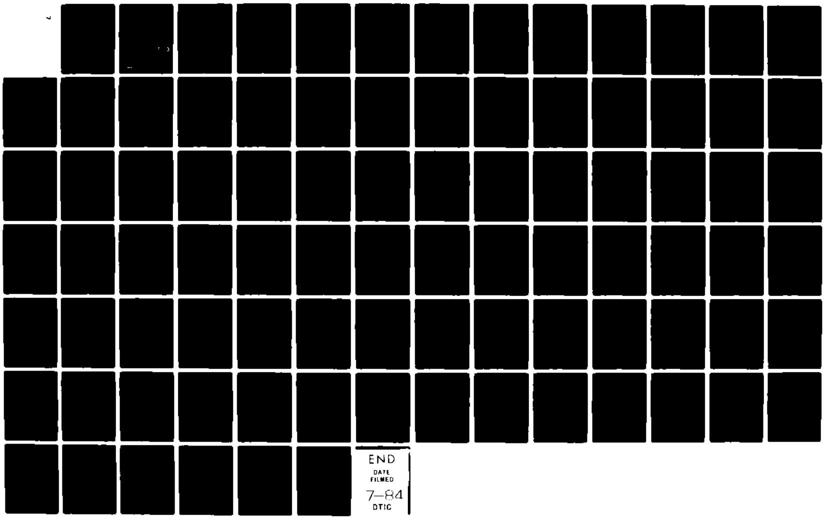
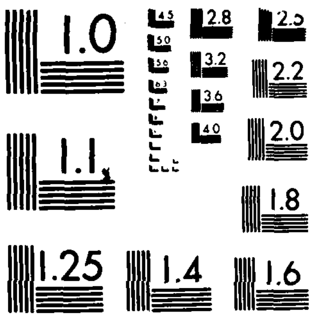
AD-A142 140 DISTRIBUTED AVIONIC COMPUTERS(U) SRI INTERNATIONAL MENLO PARK CA K N LEVITT ET AL. OCT 83 N00019-82-C-0087 1/1

UNCLASSIFIED F/G 9/2 NL



END
DATE
FILMED
7-84
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

SRI International

(12)

Distributed Avionic Computers

Final Report

October 1983

By: K.N. Levitt, P.M. Melliar-Smith, R.L. Schwartz

Computer Science Laboratory
Computer Science and Technology Divison

For:

Department of the Navy
Naval Air System Command
Washington, D.C. 20361

Attn. Mr. James Willis/AIR-310B

Contract No. N00019-82-C-0087

SRI Project 4133

SRI International

333 Ravenswood Ave. • Menlo Park, CA 94025
415 326-6200 • TWX: 910-373-2046 • Telex: 334-486

84 06 13 030

Contents

Page

Plates

Overall Introduction

The goal of the task described in this report is to establish the basis for an advanced fault-tolerant onboard computer that will be the successor to the current generation of fault-tolerant computers,(e.g., SIFT [1] and FTMP [2]). Particular features envisioned for this new computer include the following:

- Support for the processing of application programs written in a modern programming language, e.g. Ada
- Minimal burden on the programmer to prepare programs for the fault-tolerant computer
- Flexible, dynamic scheduler
- To the extent possible, an executive that can be easily ported among different processors types
- Immunity to transient faults the number of which might exceed the voting margin
- Immunity to massive transient faults, i.e., that might drive processors to a state from which they cannot proceed without the assistance of other processors
- Extendability significantly beyond that provided by SIFT

- Compatibility with the envisioned electronics system of the aircraft of the future, i.e., a large number of sensors and actuators each with its own microprocessor, and the possibility of replacing a given function that can no longer be processed by one or more backup functions.

Towards this goal we are working on the following technical problems:

a. The architecture of a network-based fault-tolerant system - Chapter 1

b. The diagnosis of transient faults from error reports - Chapter 2

c. The use of Ada as the language for the executives of the computer and the application programs - Chapter 4

d. A new paradigm (an extension of the conventional voting paradigm) for comparing the values produced by replicated processors - Chapter 5

On (a), we have identified a preliminary architecture as the basis for the research. The architecture consists of clusters interconnected by a network. Each cluster, which is logically associated with a sensor, an actuator, or a site of computation, would itself be redundant; the cluster could even be a SIFT computer whose processors are microprocessors. Different from the intracluster interconnection structure, the network that links the clusters would not be star-connected. Instead each cluster could be connected to only a few other clusters (perhaps 3). If each cluster is a SIFT (say composed of 5 processors), then the link between a pair of connected clusters could consist of 5 connections - between corresponding processors in the cluster pair. With this structure, conventional voting could be used to mask errors arising in the transmission of data between directly connected clusters. We have investigated the reliability of such a system, assuming that overall system failure occurs if any cluster exhausts its redundancy or if enough processors fail in any cluster pair such that voted communication between these clusters cannot take place.

We have also studied the problem of finding optimal network graphs. The objective here is to minimize the number of hops required for the transmission of data between any two clusters. This problem appears to be that finding low diameter graphs assuming a constraint on the fan-out for each node.

For topic (**b**), the problems are that the error report itself provides no direct information about the transient or solid nature of the fault that generated it. Such information must be deduced from the pattern and frequency of error reports. We have devised algorithms to extend those of the SIFT system, particularly to analyse interprocessor link failures, to discriminate between solid and transient faults, and to recognise solid faults with a low rate of error generation.

On (**c**), we have identified the potential advantages arising from the use of Ada. Besides the gain in portability, it is likely that the executive can be appreciably simpler than a comparable executive written in other languages (e.g., Pascal or Assembly Code) since the Ada runtime system itself provides some of the basic functions of the executive: scheduling, process synchronization, and memory allocation. The key problems we have been working on are (a) the identification of that data of the application programs that has to be voted on, (b) how to inform the executive when a vote is to take place, and (c) the identification of those points in the program where the amount of voted information is minimized. In the worst case, the amount of data to be voted on can be substantial, including: global variables, local variables, stack frames, multiprocess substructure when a task is composed of interacting subtasks, and the heap that is accessible to these subtasks. A further complication arises when the voted data has to include rendezvous information. Here interactive consistency must be used to ensure that all replicas synchronize with the same subtasks. However, the amount of data is drastically reduced by doing voting when the state of the program does not include values of the global variables, when there are not temporary variables, and when a task has no active subordinate tasks.

The assumption underlying (**d**) is that it might be advantageous to relax the principal concept underlying SIFT (and all voter-based fault-tolerant systems) that all replicas of a task get identical inputs and are expected to produce identical outputs. By relaxing this requirement it would be possible for the replicas to run at different times, thus allowing the system to be less vulnerable to correlated transient faults. If the different replicas of a task produce different values, conventional voting does not work. The voting function is replaced with a *filter* function that, similar to the conventional vote function, takes as inputs the values from the various replicas. We have started to investigate properties for this filter function. It appears that *extensions of our clock synchronization algorithm* [3] (after elimination of grossly out-of-range values, the median of the remaining values is the clock value to be synchronized to) will work for reasonably well-behaved functions. When the inputs are binary values, the function can be simpler. We do not yet have a solution when the task function does not satisfy reasonable continuity conditions.

The more theoretical aspects of this research have been supported by this contract and are reported here. Support for the more practical development of these novel architectures is being sought from NASA Langley Research Center.

Chapter 1

NETS: Network Error-Tolerant System

Introduction

Our purpose in this research is to design and assess a fault-tolerant system that could be the successor to the SIFT and FTMP class of computers. Although both SIFT and FTMP provide a reliability in the presence of permanent and transient hardware failures that far exceeds what is obtainable with conventional unreplicated computers, there remain deficiencies that appear to be inherent to the underlying architectural concepts. Among the deficiencies are:

- Limited capability for expansion beyond approximately 8 processors
- Limited capability to accommodate different processor types, including special purpose processors
- No immunity to transient faults that temporarily disable several processors

The basic problems with SIFT and FTMP are that, although they are multicomputers providing reliability through redundancy, fault-masking and logical removal of faulty processors, they employ the centralized computer technology available when the designs commenced in the 70's. In particular, they require reasonably tight synchronization among all processors and direct communication between each pair of processors. The solution to these deficiencies appears to be a more distributed concept, employing the newly available distributed system technology of the 80's.

The computer concept under consideration in this report, NETS (Network Error Tolerant System), is a bona-fide distributed system. NETS is an interconnection of *clusters*, each of which can be a simplex (nonreduntant) proces r itself a fault-tolerant computer – say a SIFT configuration of 3-5 processor NETS offers all of the "conventional" advantages of a distributed system (e.g expandability, highly-parallel computation, physical separation of computat sites), in addition to advantages particular to the goal of fault-tolerance (e.g., le s tly fault-tolerance, some immunity to massive transient faults, and better adaption to fault conditions).

NETS consists of *clusters*, each of which has a direct communication link with only a few other clusters; thus, as is standard in computer networks, communication between non-neighbor clusters requires the passing of data through intermediate clusters. Each cluster is intended to be responsible for 1 (or possibly a few) task, and is likely to be located physically close to external equipment (sensors, actuators, etc.) associated with the task. A cluster might have internal redundancy to enable it to continue operation in the presence of faults – in particular permanent or transient faults that only impact, say, 1 processor. It is anticipated that each cluster will be a SIFT configuration of 1-5 processors, 5 processors being required where higher reliability for a task is mandated and 1 processor where the task is not critical.

An initial analysis of NETS has been completed. In the process of carrying out the design, we identified a number of difficult problems, most of which we have solved at least to the point of pragmatic, if not theoretically optimal, solutions. NETS appears to address the deficiencies of SIFT and FTMP, and to achieve the high level of reliability required for aircraft electronics systems.

Section 1.2 describes in more detail the goals of an aircraft fault-tolerant system that motivated the design of NETS. The overview of the NETS architecture is presented in Section 1.3, with emphasis on the intercluster communication protocols and the requirements of the various executive-level functions. Of primary concern is failures in communication links: how to mask errors that follow such failures, how to identify faulty links, and what communication protocols can avoid the use of known faulty links. Formulas determining the reliability of NETS under various redundancy and communication assumptions are derived in Section 1.4. Section 1.5 discusses desirable properties of the interconnection network. One desirable feature is that, for a given *fan-out* **d** from each cluster and a given *diameter* **k** (the diameter being the maximum number of hops between any pair of clusters), the maximum number of clusters **n** should be accommodatable. This turns out to be a "classical" problem in graph theory, called the **(n,d,k)** problem, studied extensively at SRI and elsewhere during the 60's. We summarize the relevant results. This previous work, unfortunately, assumes no failures of communication links. Our discussion derives some initial results on the effect of faults on the diameter of certain networks.

The graphs discussed in Section 1.5 guarantee the existence of a path, between any pair of clusters, whose length does not exceed d. Algorithms for identifying the shortest path, particularly following a failure in communication link, is discussed in Section 1.6. Each cluster, being a SIFT computer, will employ our previously developed algorithm for achieving synchronization among the processors within a cluster. Extensions to that algorithm to obtain network-wide synchronization are discussed in Section 1.7. Section 1.8 discusses the related problems of recovery from massive transients and initialization of a newly connected cluster. It is shown that a cluster suffering a transient failure that corrupts data in all of the cluster's computers can be reinitialized through the efforts of the cluster's neighbors. Conditions for recovery from simultaneous massive transients are also presented. Section 1.9 presents unresolved problems and suggests experiments that could be conducted on AirLab to confirm our initial findings on NETS and to identify optimal ways of using NETS in particular applications.

1.1 Requirements of Advanced Fault-Tolerant Systems Addressed by NETS

1.1.1 High Reliability

As assumed for SIFT, the probability of a critical computation yielding an incorrect or late result is not to exceed 10^{-10}/hour over a 10 hour period. Given the current reliability of processors (even those developed using VLSI technology), this low probability of failure can be achieved only with redundancy. It is easily shown that for a given amount of redundancy, the smaller the replaceable unit the higher the reliability. This property is seen by considering a 5-fold replicated system (e.g. a centralized SIFT system). Assume the probability of failure of each processor is p, yielding a system failure probability $P_c = 5p^4$, for small values of p. This calculation assumes that a fault in a processor is detected shortly after occurrence, causing the processor to be logically removed from the configuration. Thus system failure occurs upon the 4th failure, at which time there remains one failed processor and one working processor in the configuration. In contrast, consider a highly *partitioned* system which consists of n SIFTs, each of which is 5-fold replicated, but where each of the SIFTS in this case performs $\frac{1}{n}$th of the work as compared with the original system and where the failure probability of an individual processor is $\frac{p}{n}$. For this partitioned system, the probability of failure $P_p = 5n^{-4}p^4$, or $P_c = n^4 P_p$. Of course, this simple calculation ignores the effect of any fixed, processor size-independent *overhead* associated with executive routines. Assume an overhead portion that contributes p_o to the failure probability of a processor independent of the size of the processor and also assume (very pessimistically) that $po = \frac{p}{n}$.† Then, $Pc = (2^{-4})(n^4)$. For all but very small values of n, it is seen that there is significant gain in reliability to be realized through partitioning.

†With this latter assumption, half of the computation carried out by a partitioned system processor is overhead.

1.1.2 Expandable and Contractib'ˀ

In the current SIFT architecture, each processor has a direct connection to every other processor through a broadcast link. This property limits the number of SIFT processors in a given system to about 16 - giving a range of about 3:1 from a minimal system to a maximal system. A larger range - perhaps of the order of 10:1 - would be desirable for a fault-tolerant computer to be useful for the full range of NASA applications.

On-line insertion or removal of nodes is possible in NETS. It should be possible to change the configuration without disturbing the currently proceeding computations.

1.1.3 Capability of Using Different Processor Types

One attractive feature of a network-based system is the capability to accommodate different processor types within a given system. Among the processor types could be special purpose processors (e.g., navigation computers, air data computers, etc.) in addition to general purpose processors. SIFT and FTMP, requiring tight synchronization among the processors, do not easily accommodate a wide range of processor types. Moreover, the overall reliability of the system can be improved by the use of different processor types. Our reliability computations assume processor faults occur independently. If faults are correlated, the actual reliability will be significantly lower than computed. Independence of faults is more likely if the processors have different designs and different manufacturers. Moreover, the use of different processor types will make certain software faults (e.g., in the implementation of compilers) more independent and less likely to result in system failure.

Very little special purpose hardware is required for NETS. Most of the hardware should be commercially available and known to be intrinsically reliable through extensive field use. Special purpose hardware is prone to design and, perhaps, failure in operation when exposed to unexpected environmental conditions.

Furthermore, the complexity of most specially developed chips precludes thorough testing on the part of the manufacturer. Most chip designs become reliable only after extensive testing by users followed by modification by the manufacturer. Special purpose designs will not be so thoroughly stressed and, hence, will be much less reliable.

1.1.4 Immunity to Massive Transient Faults

The current fault-tolerant systems cannot tolerate a fault that changes the values of data in a majority of the processors without causing permanent damage to the processors. Such a fault could be caused by a lightning strike or by power surge. Although not absolutely precluding global damage from massive transient faults, the physical separation of processors afforded by a network-based system should help localize the corruption caused by such faults and provide the opportunity for recovery.

1.1.5 Ability to Interface to Distributed Smart Sensors and Actuators

The trend in aircraft electronics system design is to sensors and actuators which are "smart", i.e., which provide on-site computational power. The current SIFT system interfaces with such devices through conventional Input-Output channels. A more attractive approach is to consider these devices as part of the overall network, thus enabling the more effective use of their computational power.

NETS must be designed to be capable of handling critical real-time computations. The deadlines of critical tasks must be achieved. The interaction of unpredictable tasks compounds the difficulty of demonstrating that task deadlines are satisfied in a distributed system.

As described in the following sections, the NETS architecture can satisfy all of the above goals.

1.2 Overview of NETS Architecture

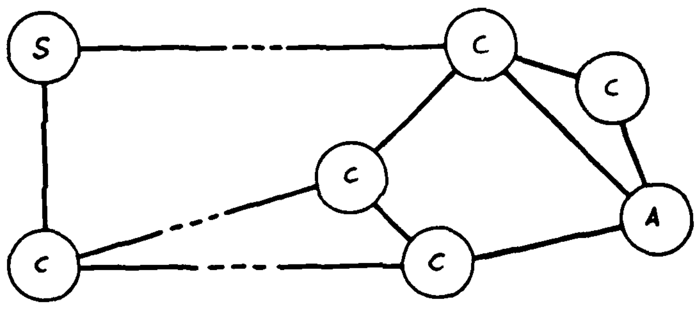
This section presents a brief overview to the NETS architecture. Issues covered are

- The design of clusters that comprise the nodes of NETS and possible protocols for intercluster communication, including the accommodation to faulty links
- The combination of clusters of different redundancy
- Requirements for an overall network executive that is distributed among the nodes of NETS

1.2.1 Clusters and Their Communication Protocols

The computational unit in NETS is called a *cluster*; the clusters communicate with each other through a network. As illustrated in Figure 1, a cluster is a site that can be associated with a sensor, an actuator, or can be a computation cluster whose role is to generate outputs in response to inputs. A *sensor* cluster will have no logical inputs, and an *actuator* cluster no logical outputs. A *computation* cluster will have both logical inputs and logical outputs. The interconnection network will not be a *complete* graph; that is, each cluster will not have a direct connection to every other cluster. Hence intermediate hops will be required when a pair of nonadjacent clusters communicate with each other.

In generating the value to be delivered to an actuator in response to sensor inputs, all three types of clusters could be involved. A chain of tasks (in general a tree if it is assumed that more than one sensor is involved) cooperate to generate the output, the sensors providing the inputs, the computation clusters generating intermediate values, and the actuator generating the final value. In its most general form, the graph of these clusters will be as indicated in Figure 2. The sensors and actuator clusters are "stubs" hanging off a general graph (containing loops). The loops are present to account for one (or more) computation cluster executing more than one task in the chain of tasks.

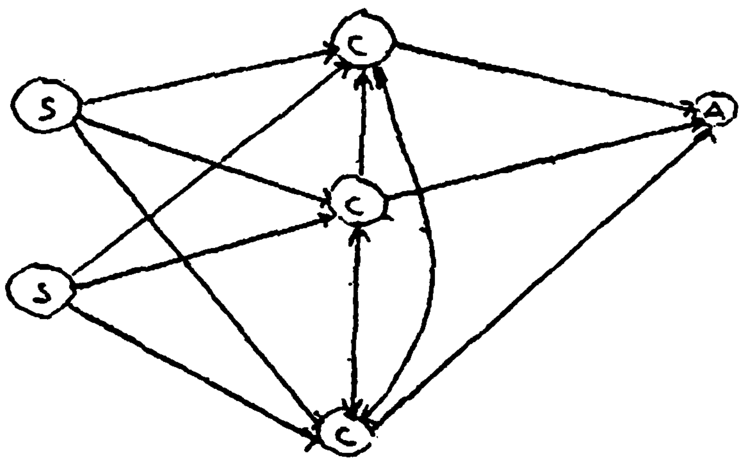


S: Sensor Cluster

A: Actuator Cluster

C: Computation Cluster

Figure 1. NETS is an Incomplete Interconnection of Clusters



Temporary Figure

Figure 2. The General Form of the Cluster Graph

Each cluster is configured as a SIFT computer, i.e., a complete interconnection among a set of processors. It is expected that a given NETS system will have clusters of different size; we call such a cluster an *n-SIFT*. We will represent the internal structure of a n-SIFT as the schematic illustrated (for n=5) in Figure 3. It is likely that the maximum value of n needed for currently envisioned applications is 5, as the probability of failure of a 5-processor SIFT is quite low. Of course as discussed below, for less critical tasks clusters containing fewer than 5 processors will suffice.

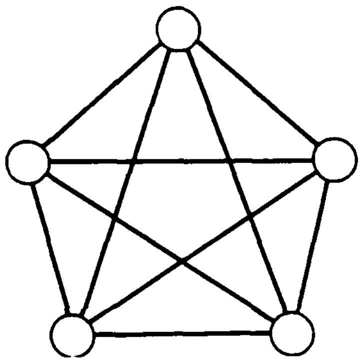
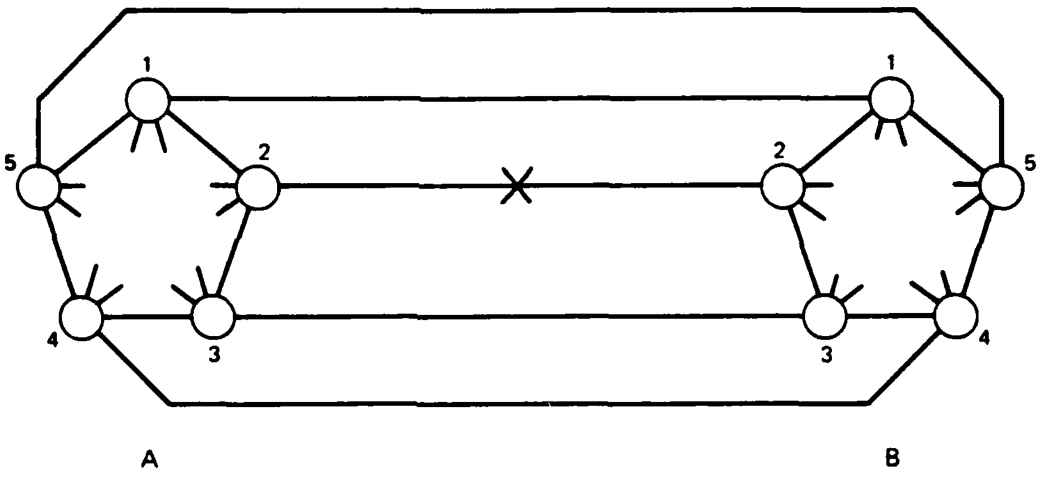


Figure 3. Within a NETS Cluster the Interconnection is Complete

Let us now consider the structure of the interconnection between a pair of clusters. As illustrated in Figure 4 for the two 5-SIFTs, there is a single link between corresponding processors; for the current discussion let us assume that the link is bidirectional, although unidirectional links are possible. Assume that a task a executing on cluster A is required to transmit data to a task b executing on B. In the absence of failures, each A processor will send the data to its corresponding B processor. When all of the B processors have received the data, they exchange the received values and vote. What if a link suffers a failure? Since a link is just a wire connecting a pair of processors, it is convenient to view a link failure as a failure in *either* of its associated processors. Accordingly, an error resulting from a link failure can be masked as long as there is adequate voting margin; what constitutes an adequate voting margin for link failures is discussed in the next section. Furthermore, once the link failure is identified, that link can be avoided in future communication between A and B. The identification and reporting of link failures is quite straightforward, since it is simply the identification and reporting of processor failures. Thus the link (A-2, B-2) will be assumed to be faulty under any of the following conditions:

a. The processors of cluster B (A) determine processor B-2 (A-2) to have suffered a permanent fault through B-2 (A-2) being outvoted on some computation.

b. B-2 (A-2) reports itself to be faulty to more than one of its neighbors in B (A).



Link 2-2 fails if A2 or B2 fail

Figure 4. Interconnection between two 5-SIFTs

Note that in producing erroneous outputs B-2 (A-2) could be faulty itself or could have received erroneous data from the other cluster - A-2 or B-2. The *safe* policy is to assume *both* A-2 and B-2 are faulty. Note, however, that a link could fail but the associated processors could still be capable of performing other activities, e.g., compute on behalf of tasks or transmit data along other links (see below). Hence a policy less profligate in dismissing processors would be as follows:

If a processor (say, B-2) is outvoted on data received from A, assume the failure could be either in A-2 or B-2 pending confirmation by subsequent error reports. That is, if no reports are received indicating that A-2 is unable to carry out its task processing activities, it is allowed to participate in all activities of A except the transmittal of data to B. Alternatively, future reports could indicate that B is likely to be working.

Once it has been determined, by either A or B, that a link is faulty, the cluster noting the failure informs its neighbor that the link is to be avoided. The direct exchange of failure information is possible if the links between processors are bidirectional; otherwise, as discussed below, the information must be transmitted through a path that contains other clusters.

The discussion above is concerned with the case where cluster B contains a task that requires data from its neighbor A. What if the destination of A's data is to be a third cluster C which is not a neighbor of A? For example, the transmission might require an intermediate hop through B. In this case, the working processors of A transmits the data to B using working links The working processors of B vote on the received values and then transmit the voted values to C, again using working links. For example, as illustrated in Figure 5, A would avoid the link (A-2,B-2) and B the link (B-3,C-3), assuming these links were known to be faulty. By voting on the data received from A, B can mask any errors from newly failed links between A and B, thus increasing the chances for transmittal of error-free data to C. In addition, B can immediately take note of a failed link and inform A of the failure. We call this approach the ***vote and forward*** protocol. Through this protocol, errors are handled by the cooperation of the two processors connected by the failed link; no other clusters need participate. The disadvantage of the vote and forward protocol is the delay it introduces.

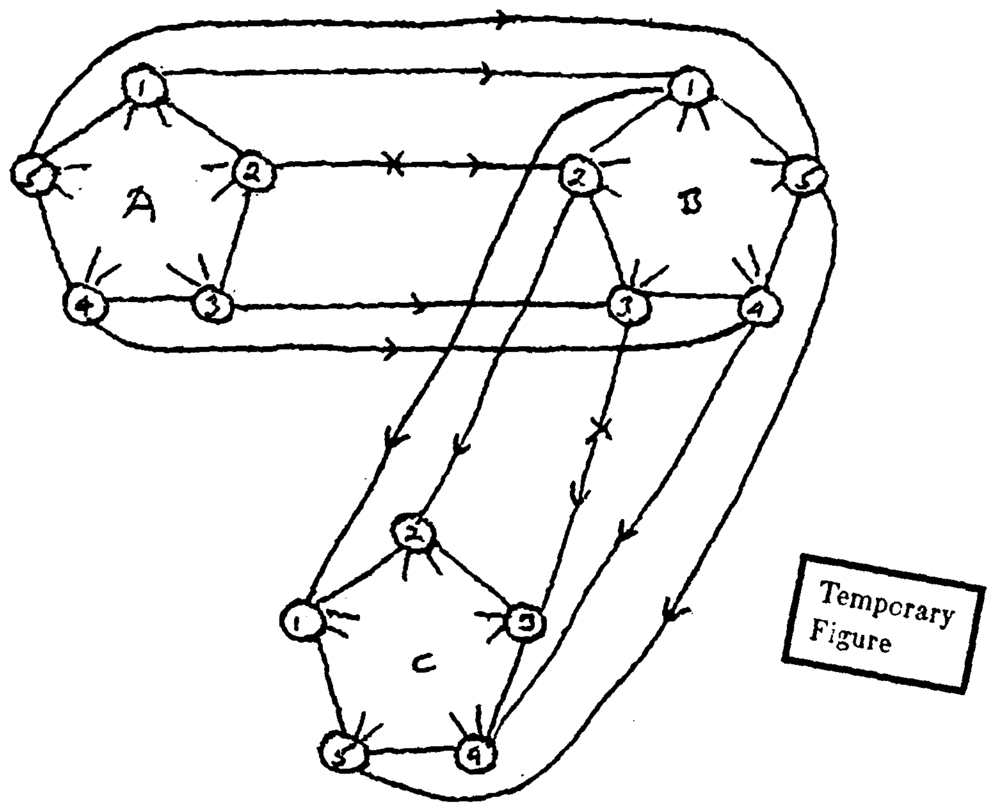


Figure 5. Use of an Intermediate Hop to Transmit Data from a Source to a Destination

A different protocol, which involves less delay, is called the ***forward and vote at destination*** protocol. Data received by an intermediate cluster is forwarded to the next cluster on the path ***without*** voting. Once all of the replicas arrive at the the destination cluster, they are voted on. The successful masking of errors requires that the paths taken by the different replicas be ***nonoverlapping***, otherwise a single link failure could cause correlated errors. A further complication is in locating faulty links, as a single error report can only locate the error to a path which might contain a number of links. It would be necessary to exercise, subsequently, each of the suspected links to try to locate the faulty link.

1.2.2 The Combining of Clusters of Different Redundancy

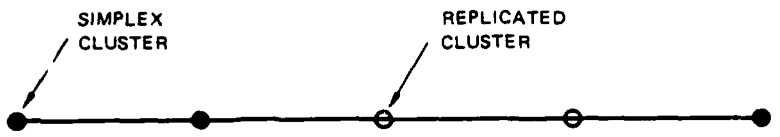
One of the goals for NETS that we indicated in Section 2 is the capability to handle tasks of different criticality without enduring the penalty of excessive redundancy for the less critical tasks. NETS can achieve this goal through the use of clusters containing different number of processors. Highly critical tasks would be assigned to clusters containing 5 processors; less critical tasks to 3-processor clusters; uncritical tasks to 1-processor clusters.

If clusters containing different levels of redundancy are to be combined in a single NETS, one approach is to segregate the clusters of a given redundancy to their own subnetworks of NETS. However, it is possible to combine clusters of different redundancy without compromising the reliability goals.

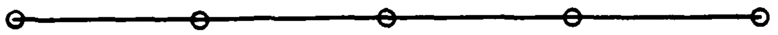
What are acceptable communication paths between clusters, particularly if the paths might involve clusters whose redundancies are not the same? Let us consider a source cluster A sending data to a destination cluster B through a path containing other clusters. Assume that A and B have the same redundancy n. If the *forward and vote at destination* protocol is used, the ideal is that the number of distinct paths from source to destination be equal to n. The minimum requirement is that at least 3 distinct paths be used to protect against any single point failure.

If the *vote and forward* protocol is used, all clusters on the path should ideally have the same redundancy as that of the source and destination clusters. Figure 6 summarizes the various possibilities.

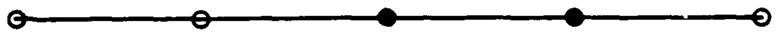
One additional issue in combining clusters of different redundancy is balancing the load on each processor. That is, if m 3-SIFTS are to be neighbors of a 5-SIFT, what should the interconnection pattern be so that each of the processors of the 5-SIFT have approximately the same fan-out? The solution is quite simple, and is best illustrated through example.



(a) Acceptable Path
End nodes - simplex
Intermediate nodes - simplex or replicated



(b) Acceptable Path
All nodes replicated



(c) Unacceptable Path
End nodes - replicated
Intermediate nodes - simplex or replicated

Figure 6. Communication Paths containing Simplex and Replicated Clusters

Assume that the number of external links to each of the 5-SIFT processors is not to exceed 2. Then, as illustrated in Figure 7, three 3-SIFTS can be neighbors of the 5-SIFT using the interconnection pattern indicated. Each of the processors of the 5-SIFT, except 2, have a fan-out of 2. The interconnection pattern can be described using a matrix notation, as below.

	Processors of 5-SIFT				
3-SIFTs	1	2	3	4	5
A	x	x	x		
B	x			x	x
C		x		x	x

According to the matrix, the 3 processors of A are connected to processors 1,2, and 3 of the 5-SIFT respectively, etc.

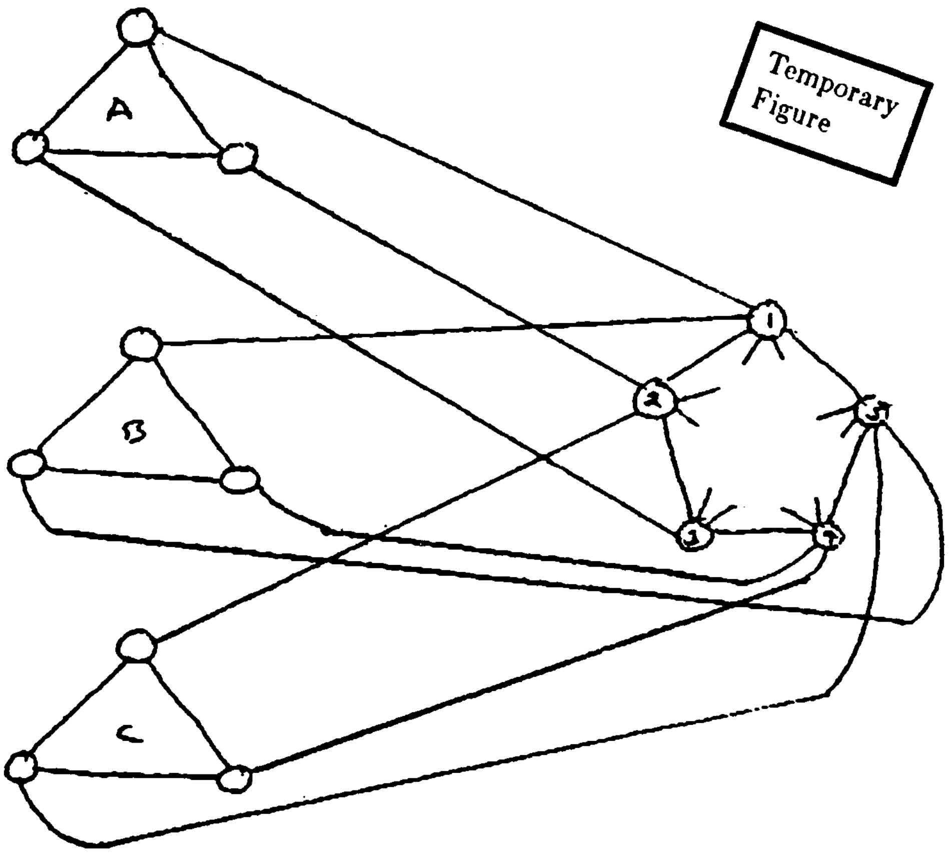


Figure 7. A Balanced Connection of 3-SIFTs with a 5-SIFT

In general, if the allowed fan-out from each of the 5-SIFT processors is d, then the maximum number of 3-SIFTs that can be connected is given by the quotient $\frac{5*d}{3}$. Thus for d=3, the following matrix will apply.

	Processors of 5-SIFT				
3-SIFTs	1	2	3	4	5
A	x	x	x		
B	x			x	x
C		x	x		x
D			x	x	x
E		x		x	x

Note that the data passed to a 5-SIFT from a 3-SIFT will be subjected to only 3-way voting.

1.2.3 Requirements of NETS Executives

It is envisioned that each of the clusters of NETS will run the SIFT executive: local executive, error report, global executive, etc. To manage the network itself, a **network executive** is required. The network executive functions, distributed among the clusters, are the following:

- Apply the vote and forward protocol to messages destined for other clusters. It is envisioned that each message will have a destination tag, and each cluster will have a table indicating which neighbor to use for each possible ultimate destination.

- Receive and process error reports from neighboring clusters. The processing will identify faulty links, which are to be avoided in subsequent communications. The identity of a suspected faulty link is broadcasted to the cluster at the other end of the link.

- Determine *optimal* paths to be used in the communication of data between clusters, where optimal means shortest. As discussed in Section 6, this function will be invoked when enough links between a pair of clusters have failed, thus precluding the reliable communication between these clusters. Many communication paths might have to be changed. As we show, the determination of new paths can be carried out *locally* in the sense that each cluster decides on the new optimal paths from information received from its neighbors.
- Participate in the initialization of newly connected neighboring clusters and in the recovery of neighbors that have suffered massive transients that temporary disable a majority of the cluster's processors. The approach to both of these problems is discussed in Section 8.

1.3 Reliability Assessment

In this section we consider the reliability achievable by NETS. We consider the following failure modes: (1) permanent faults - system failure due to exhaustion of spares, and (2) permanent faults - system failure due to buildup of faults beyond the voting margin before reconfiguration is completed. For each of the modes we consider separately the cases of (a) cluster failure, preventing it from performing tasks; and (b) link failures, preventing a cluster from communicating with *any* of its neighbors.

It is shown that acceptable reliability - better than the basic requirement of 10^{-10}/hour for critical tasks - can be obtained, even for relatively large NETS systems, assuming that (1) all critical tasks are executed on 5-SIFT clusters, (2) all communication between critical tasks is through 5-SIFT clusters using the vote and forward protocol, and (3) the fan-out from each cluster is at least 2. It is encouraging to observe that such a modest fan-out is acceptable; other requirements (e.g., keeping the communicating paths short) will probably force us to consider a higher fan-out.

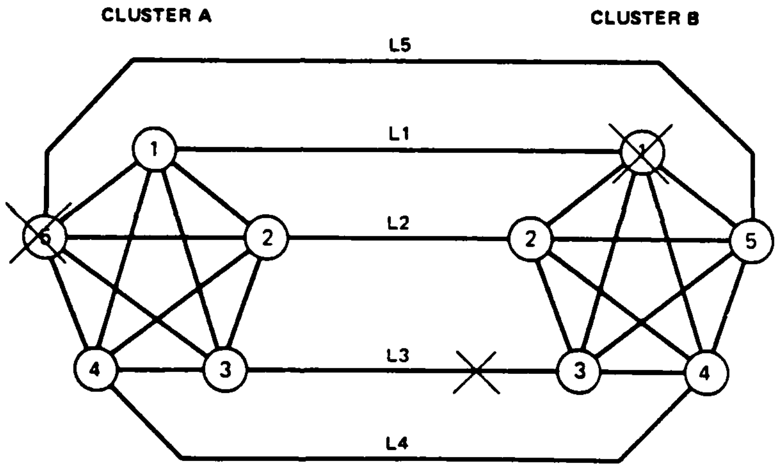
1.3.1 System Failure Due to Exhaustion of Spares

We consider a NETS system to be a network in which each node is a 5-SIFT. (Clearly, we do not imply that all of the clusters must be 5-SIFTS; Our intention is to derive a lower bound on the reliability that critical tasks would experience.)

Let us assume, for this section, that faults become detectable errors that are handled very shortly after their occurrence. Thus a cluster will continue working through its third failure, as two working processors remain. However, the next fault spells the failure of the cluster, as one good and one bad processor would remain. Thus the probability of 4 (out of 5) processor failures is approximately $5p^4$, where p is the probability of a cluster failure. (Again, we are assuming faults occur independently of each other.) The probability of a failure of one cluster in an N-cluster system is then $5Np^4$.

Now let us consider the probability of system failure due to a sufficient number of link failures occurring such that a cluster cannot communicate with any of its neighbors. We will only be enumerating those failure conditions that do *not* constitute cluster failure. Our initial assumptions will be as follows: (1) fan-out of two from each cluster, and (2) the forward and vote protocol; later we consider higher values of fan-out (which will provide improved reliability and the forward and vote at destination protocol.

1.3.2 Two-Cluster System

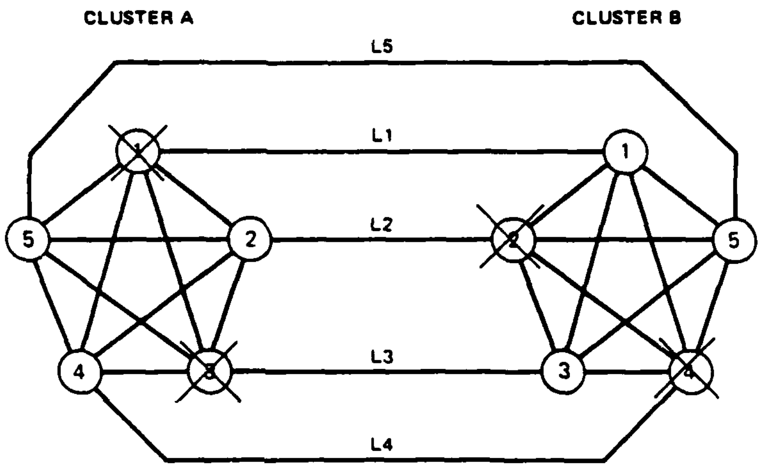


After handling failures in B1 and A5, the system can tolerate failure of L3.

Figure 8. Failures tolerated in Two Cluster Nets.

First we will show that *all* patterns of three link failures can be tolerated. Next the number of patterns of 4 link failures leading to system failure will be enumerated. Then the probability of system failure will be approximated as the sum of the probabilities of link failures that cause system failure and cluster failures that cause system failure.

We assume that link failures are detected immediately after their occurrence. Thus, referring to Figure 8, failures of processors B1 and A5, implying failures of links L1 and L5, would be detected in turn and handled by the **network executive**. At this point, there are 3 working links L2, L3, and L4, allowing the system to tolerate another link failure – say L3; the results emerging from the two working links would outvote the result from the newly failing link.



Examples of untolerated patterns of 4 failures are:
A1,A3,B2,B4 A1,A2,A3,B4 and A1,A2,A3,A4.

Figure 9. Failures not tolerated in Two Cluster NETS.

Now, as depicted in Figure 9, let us consider patterns of 4 faults that lead to link failure causing system failure. We claim that faults in A1, A3, B2, B4 is such a fault pattern, as it implies failure of four links: L1, L2, L3, and L4. (Note that not all patterns of 4 faults lead to system failure; for example, faults in A1, A2, B1, and B2 would cause only 2 links – L1 and L2 – to fail.) Another pattern of 4 faults that leads to system failure is A1, A2, A3, and B4. It is easily shown that there are two classes of failures to be considered. For each of these classes communication between the clusters can no longer be guaranteed, although there are still adequate resources left in each of the clusters to allow voting to mask all internal cluster failures.

1. Failures of processors Ai, Aj, Bk, Bl, where i, j, k, l are all different (as illustrated in Fig. 6). Only one reliabl^ link now exists between the two clusters (link 5 in the figure), in which case the voting of results transmitted between the two clusters no longer masks errors. The number of such patterns is $\binom{5}{2} \times \binom{3}{2} = 30$, where $\binom{n}{m}$ is the *combination* function – the number of combinations of n items taken m at a time.

2. Failures of processors Ai,Aj,Ak,Bl or Bi,Bj,Bk,Al, where i, j, k, l, are all different. Similar to the situation in (1), only 1 reliable link (link 5) remains for intercluster communication. The enumeration here yields $2 \times \binom{5}{1} \times \binom{4}{3} =$ 40 failure patterns.

(Note that a failure of 4 processors all within a cluster is not considered here as it implies a failure of the cluster itself – see below) Summing (1) and (2) yields 70 failure patterns or a failure probability of approximately $70p^4$, where p is the probability of failure of an individual processor.

A cluster itself fails when the number of operational processors within a cluster is inadequate to permit error masking through voting. Assuming, as above, that faults occur at a low enough rate to permit the logical removal of a faulty processor before the occurrence of a subesquent fault, failure of a cluster occurs when 4 faults are occur. The probability of 4 faults within a cluster is approximately $\binom{5}{4} \times p^4$ or $10p^4$ when both A and B clusters are considered. Thus the probability of system failure due to link failure and the probability of system failure due to cluster failure are approximately the same.

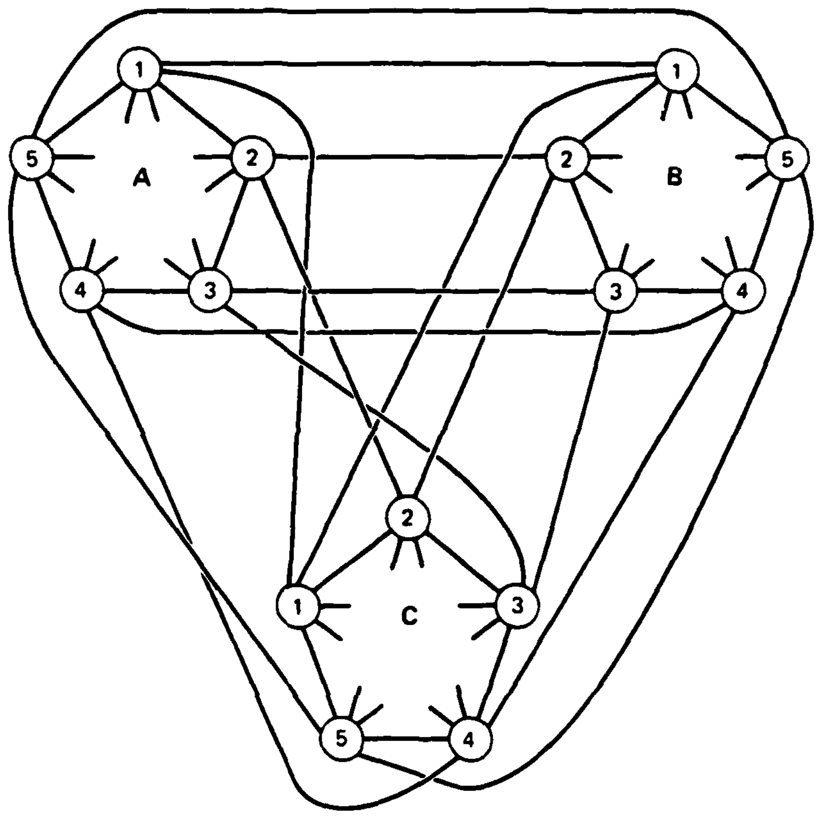


Figure 10. A Three Cluster NETS

1.3.3 Three Cluster System

Figure 10 depicts a 3-cluster NETS system; the fan-out from each cluster is two. As in the 2-cluster system, all patterns of three link failures are tolerated, but here, with the use of routing of broadcasts via intermediate cluster, all patterns of four link failures are also tolerated.

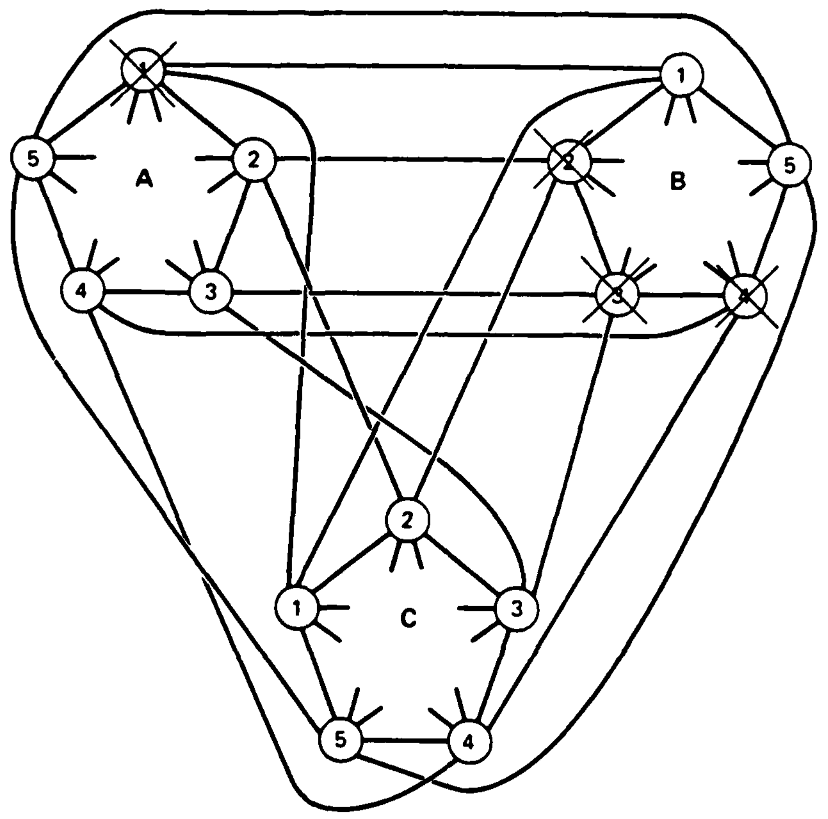


Figure 11. A pattern of Four Faults that does not cause Link Failure

Figure 11 illustrates the probability that a fault pattern occurs that prevents A from communicating with either of its neighbors. A cluster I will be unable to communicate directly with cluster J, if 4 (or more) of the links connecting I and J are failed. Note that if 3 or fewer links have failed, the voting margin suffices to allow reliable communication under the assumption that a faulty link is ignored prior to the next occurrence of a faulty link. We first observe that all patterns of 4 failures spread over these 3 clusters are tolerated. (The reader is reminded that we are excluding from the enumeration those failure patterns that cause cluster failure, e.g., the failure of 4 processors in A.) An example of a pattern of 4 such tolerated failures is shown in Figure 11. The four failures indicated would prevent A from communicating directly with B, as only one good link (A-5,B-5) remains between these two clusters. However, 4 good links remain between A and C, and 3 good links remain between B and C, thus allowing communication between A and B to be through C. The other patterns of 4 faults among the 3 clusters are: two in each of two clusters, two in one cluster and one each in the other two clusters - all of which can be shown to be tolerated.

Now consider the patterns of 5 failures that are not tolerated. One such pattern, as shown in Figure 12, prevents A from communicating with either B or C - in effect isolating A. It can be shown that there is no pattern of five faults two of which are in A such that A will not be able to communicate with *either* B or C. Hence the only case of interest is three faults in A. If the two remaining faults are both in the same cluster, then A will have two working links on which it can communicate with the other cluster, thus avoiding isolation. Hence the enumeration need only consider three faults in A, one fault in B such that A cannot communicate with B, and one fault in C such that A cannot communicate with C. The number of such patterns is given by

$$C(5,3) * C(2,1) * C(2,1) = 40.$$

Then, an upper bound on the probability of system failure due to link faults is $40Np^5$. This is an upper since some of the patterns covered for a cluster will also spell failure for a neighbor.

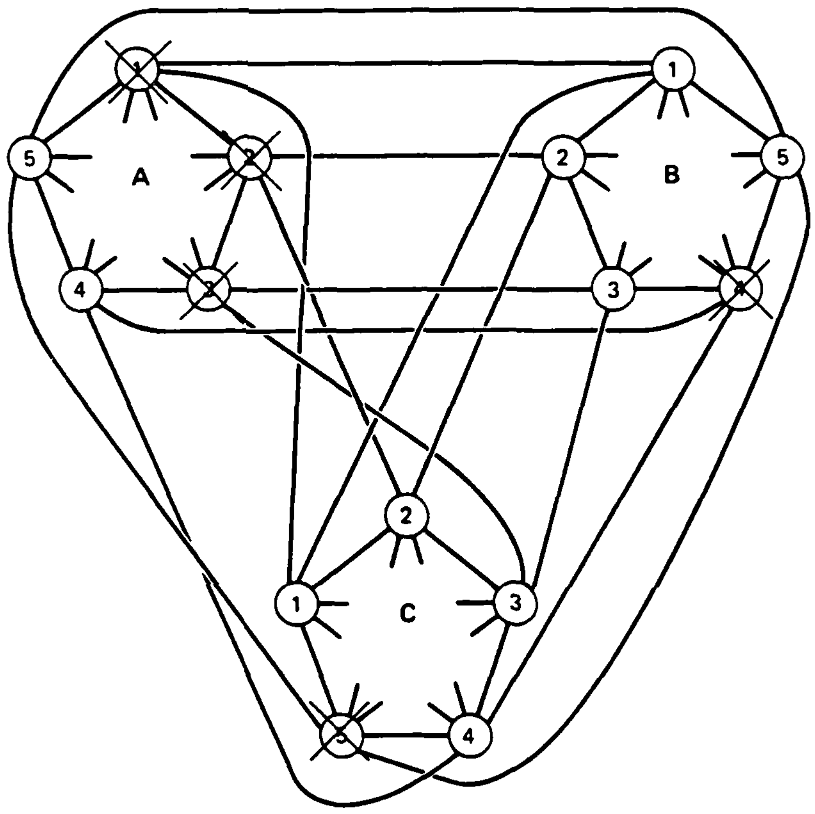


Figure 12. A Pattern of Five Failures that is not Tolerated

Note that for reasonably small values of p, the probability of system failure is dominated by the probability of cluster failure. Further note that the probability of A not being able to talk with *both* of its neighbors is given by $100p^4$ (enumerating the patterns of four faults that cause either intercluster communication path to become unreliable). Exceeding the probability of cluster failure by a factor of 20, this is probably too high for critical computations. Thus it is necessary to allow for alternate communication paths in NETS.

If the intercluster fan-out is increased to 3, the probability of system failure due to link failure is decreased to $60Np^6$. Again, it is probably not necessary to employ a fan-out of 3 from the standpoint of achieving reliable communication.

Now let us consider the use of the forward and vote at destination protocol. All that is required is that A have two or more working links - both to the same neighbor or one one to each of its neighbors. In this case all patterns of 5 failures are tolerated, but at the expense of the more complicated protocol.

1.3.4 General Result for N-Cluster Systems with Fanouts of 2 or 3

Consider a fan-out of 2 from each cluster. System failure will occur when one (or more) clusters is unable to communicate with any neighbor, thus isolating it from the rest of the system. Enumerating the faults that cause this situation, we find that the probability of system failure due to link faults is given by $N \times 40 \times p^5$. (This result is easily derived as a generalization of the enumeration carried out for the the 3 cluster NETS.) The probability of system failure due to cluster failure is $N \times 5 \times p^4$; thus, for this general case, the fan-out of 2 is adequate for achieving reliability.

If the intercluster fan-out is increased to 3, the probability of system failure due to link failure is decreased to $N \times \binom{5}{3}\binom{2}{1}^3 \times p^6 = N \times 80 \times p^6$.

1.3.5 System Failure due to Fault Buildup Prior to Reconfiguration

Unlike the adaptive voting approach assumed in the previous section, we are assuming here that faults are not detected and handled. Thus system failure will occur whenever three bad inputs are generated - either within a cluster carrying out a computation or in the passing of data between clusters. The probability of three faults in a 5-SIFT is given by $10p^3$.

On the other hand, the communication between a pair of clusters (A and B) will become unreliable when A suffers 2 faults (say in Aj and Ai) and B suffers one fault in Bl, l =j,i. The number of such fault patterns is

$$C(5,2) * C(3,1) = 30.$$

Thus the probability of system failure due to cluster failure and that due to link failures are comparable. Furthermore, there is no alternative to improving the reliability in this case short of increasing the redundancy level of the clusters.

1.4 Structure of the Interconnection Network

In this section we consider the interconnection network through which clusters communicate with each other. One key property of the network is that it allow clusters to communicate with each other with minimal delay. For the moment let us assume that each cluster has the need to communicate with every other cluster. (It is understood that this assumption ignores the possibility of assigning collections of tasks that communicate with each to collections of clusters that are close to each other; this possibility is discussed later.) Hence a measure of the quality of a network is the ***diameter*** k of the network. Here diameter is taken in the graph-theoretic sense to mean the following:

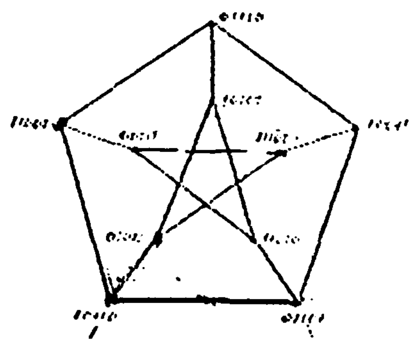
> Let the distance between any pair of adjacent nodes be 1. Let the distance between any pair of nonadjacent nodes i,j, be the length lij of the shortest path between i and j. The diameter k of the graph is maximum shortest path, where the maximum is taken over every pair of nodes. Thus for a diameter k graph it is assured that no more than k hops need be taken in going between any pair of nodes.

As might be expected, the diameter of a graph generally increases with the fan-out d permitted from each node. In the limit, if every node is connected to every other node, the diameter is one. However, we are seeking graphs in which the fan-out is much less than the number of nodes. In this case, a more comprehensive measure of the quality of the graph is the number of nodes n, for a given d and k, the general desire being to find graphs with maximum n. A graph having n nodes, diameter k, and fan-out d is called an *(n,d,k)* graph. A graph having the largest n for given d and k is called an *(n,d,k)max* graph.

One further complication is the impact of faulty links. We want the diameter to be low despite the occurrence of faulty links - say t such faults; whenever a fault occurs it is necessary to find a new shortest path that does not include the faulty link. A graph of n nodes, fan-out d, and diameter k in the presence of k or fewer faulty links is called an *(n,d,k,t)* graph. Again, we are, in general, interested in maximizing n for fixed values of the other parameters - leading to *(n,d,k,t)max* graphs.

During the 60's, a number of researchers searched for (n,d,k) graphs, in particular for families of such graphs covering different values of n,d, and k. Unfortunately, all of this work was aimed at the fault-free case. Below we discuss several such families and show how the diameter is affected by a single link failure occurring anywhere in the graph - the case t=1. As discussed in the previous section, the occurrence of more than one link failure is so unlikely that we need not consider it.

For one particularly interesting family - due to Akers [?] and shown if Figure 13 - we show that the effect of a single fault is to increase the diameter by no more than 2 for graphs at the low end of the family - containing up to 35 nodes. For graphs at the higher end, the single link failure will not cause the diameter to be increased at all. Even for those graphs where the diameter is increased, for most node pairs, the shortest path is not increased due to the failure. However, more work is needed here, particularly if optimum graphs are to found.



Temporary Figure

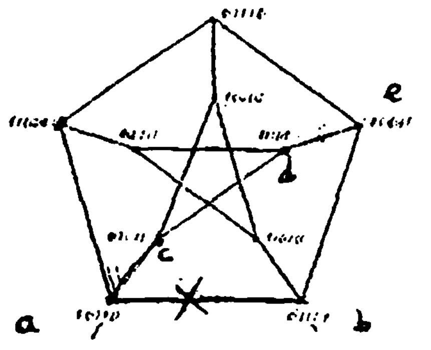
Figure 13. An Akers (n,d,k) Graph

Let us briefly consider the fault-free case. Moore [?] has computed an upper bound on n for given d and k. The reasoning is as follows. Consider the maximum number of nodes in a graph such that the distance from *one distinguished* node a to *any* other node is no more than k, assuming a fan-out of d. Let a be the root node of a tree. Let there be d successors to a, as allowed by the fan-out limitation. Let each of these successors have d-1 successors, and each of d words per cluster. Approaches similar to this are used in most networks [?].

Can this approach be easily extended to to faulty link case? The trivial extension is to store for each i triples <j,k,l>, which would indicate the successor j on the shortest path to k for link l being faulty. For a network of 500 links - a reasonable number of links for 100 or more nodes - the storage requirement could increase to 50,000 words. This, then, is not an effective solution. Some savings can be attained, however, if an Akers' graph is used. Since the alternate paths do not overlap with the "original" path, the third position in the triple can be a node instead of an edge, the intent being to avoid that node. The storage would then be reduced to, perhaps, 10,000 words. A further reduction can be obtained by only storing triples for those paths that are changed by a fault. We believe, although have not verified, that only about 20%of a graphs shortest paths are affected by any particular link fault. Thus the storage requirement might be as low as 2,000 words. This would probably be acceptable.

Nevertheless, there are techniques for dynamically determining the shortest path that involve computation, but at a significant savings in storage. The following is one such approach, which we conjecture to find the alternate shortest path or determine that the primary path is still applicable.

Each node i will have a table of triples <j,j',k>. As before, j is the primary successor to i in communicating with k. However, an alternate j' is now also specified. Each node j will also have *back pointer* pairs <i,k>, indicating that j is on the shortest path from j's neighbor i to k. The key is to determine if the occurrence of a fault requires a shift to the alternate. Consider Figure 14. Assume the site of the fault is the link a-b, between processors a and b. Also assume that the occurrence of a fault but no. its identity is broadcasted around the net by either a or b. The key, now, is to determine which primary paths involve a-b; for those that do, there will be a shift to the alternative. Nodes a and b, by referring to their tables, for which destinations k the link a-b was on the primary path. They indicate this information to each of their neighbors, which switch to the alternate successor if their primary successor was indicated by a or b to be affected. These neighbors n of a and b then broadcast to their successors those destinations for which any member n required a shift to an alternate path. The process continues until all nodes have received updates. As presented the process proceeds in synchronized steps starting with a and b. However, the process need not synchronized; each node upon receiving *any* information updates its tables and broadcasts derived information to its neighbors. All that is required is enough time for the process to yield no new information.



Temporary Figure

Link a-b is faulty

1. Alternative path between a,b is (a,c,d,e,b) and has length 4
2. Alternative path between c,b is (c,d,e,b) and has length 3

Figure 14. An Akers Graph with a Single Link Failure

1.5 Synchronization in NETS

The successful operation of a SIFT cluster requires that its processors be synchronized to within approximately 50 microseconds. Such synchronization is necessary to prevent a processor from changing is rate of processing tasks to the point where it is working on an iteration that is different from its neighbors, and thus producing different results and destroying the exact match required for voting.

In NETS, of course, each SIFT cluster would have to be internally synchronized. However, the clusters need not be synchronized with each other. The internal synchronization will guarantee that the processors of each cluster produce data for output at approximately the same time, and that the processors of a cluster reading this data will all read it at approximately the same time. The only danger is that a cluster might run faster to the point where an occasional iteration of data is not read by an input cluster before being overwritten by the next iteration's data. We do not see this as a problem, as long as each cluster continues to work with the most recent data available. Moreover, it might be difficult to synchronize the clusters of NETS if they have different type processors with different clock rates.

Nevertheless, there might be occasions where it is desirable to have the clusters in NETS synchronized with each other. If each cluster contains at least 4 processors and if the fan-out from each cluster is 3, then the current SIFT synchronization algorithm can be used by each cluster to synchronize itself with its neighbors.

It should be noted that the algorithm can also be used for clusters containing fewer than 4 processors. In this case, each processor will synchronize itself with its neighbors within a cluster *and* in other clusters.

1.6 Recovery from Massive Transients

We assume the occurrence of a fault that corrupts the state of every processor in a cluster, but does not cause damage to prevents its subsequent processing. This kind of fault we call a ***massive transient*** fault. A cluster c suffering such a fault may not be able to effect recovery without outside assistance. For example, the information indicating working processors and, perhaps more fundamentally, indicating the neighbors of c might be have been lost due to the fault. The following are requirements to permit the recovery of a cluster through the assistance of its neighbors:

- Restart Box (rb)
- Checkpointing of global data
- Detection of Massive Transient
- Recovery process

We will require a modest-size special purpose circuit in each cluster, which we call a *restart box* (rb). A rb will accept inputs from each of c's neighbors, requesting c to return to a reset state. To prevent a neighbor, perhaps one that itself has suffered a massive transient, from maliciously trying to restart its neighbors, restart will only be carried out if at least 2 of c's neighbors send restart signals. The immediate effect of being in the reset state is to execute the clock synchronization n algorithm and run an initialization program that will continue recovery (see below).

Certain critical data of a cluster must be checkpointed. This includes the identity of the working processors, the identity of neighbors (assuming such information is not hard-wired), the pairs and triples needed to communicate with other clusters on shortest paths, and the identity of which links with neighbors are working. Such information can be given to a neighbor each time it is updated. Not required to be checkpointed would be task data (it is regenerated each iteration) and task schedules (they are likely to be stored in microcode).

When a cluster has suffered a massive transient it is assumed that its behavior becomes erratic. This could involve the processors becoming unsynchronized, the loss of data such that there is little agreement among the output values of the cluster's processors, or the absence of any output data. Any of these events, when observed by a neighbor, are evidence of a massive transient. In any event, a cluster exhibiting behavior cannot produce any useful work. It would also be possible for a neighbor to submit test data and, based on the return, decide that the cluster has suffered a massive transient.

The *recovery* of a cluster is as follows. It must be given all of the data it previously checkpointed and be moved to a state where it starts to execute the tasks on its schedule. The data will come from its neighbors, once the cluster indicates that it is in its reset state. The final input from the neighbors will move the cluster to the state where it commences doing useful work.

1.7 Remaining Problems and Recommendations

We believe that the work conducted to date demonstrates the feasibility of the NETS concept for the aircraft environment. Some additional problems, solution to which would optimize the NETS design are the following:

- Transient Fault Analysis: A transient fault causes a processor to temporarily deliver erroneous results. After a period of time, the processor will return to a state where it will deliver correct results. The usual technique for dealing with transient faults is permit the processor suffering the fault to deliver the erroneous values for a certain period of time t, during which voting will mask the error. If the processor does not return to an error-free state within t, it is considered to have suffered a permanent fault. If t is set at too low a value, then long-duration transient faults will be considered as permanent faults, and good processors will be removed from the system. On the other hand, if t is set too high, transient (and permanent) faults can build up, causing the system to fail by having the voting margin exceeded. To determine the vulnerability of NETS to transient faults, it will be necessary to weigh the probability of exceeding the voting margin against the probability of running out of spares.

- Determination of Near-Optimal Interconnection Networks. Based on the work carried out in the 60's, adequate <n,d,k> graphs are known. However, the situation is not as encouraging when faults must be handled - the <n,d,k,t> case. We have shown that the Akers' graphs have good fault-handling capabilities - at least for the case of single link faults. It is recommended that other families of graphs be sought. Moreover, reasonably tight upper bounds on n should be determined to guide the search for such graphs.

- A Distributed Algorithm for Computing Optimal Communication Paths: It will be necessary to associate paths in the interconnection graph with each pair of clusters that communicate with each other. The computation of such optimal paths can be done "offline" for the initial configuration. As link failures occur, certain paths might become closed off, in which case it will be necessary to determine new paths. It is conjectured that this determination of optimal paths can be accomplished in a local manner as follows: If cluster A can no longer use B in communicating with C, it chooses to communicate with C using that neighboring cluster D such that the distance between D and C is smaller than the distance between any other neighbor and C. It remains to verify this conjecture. Also, we must consider the impact of a second failure while the new shortest path is being computed.

- Assignment of Tasks to Clusters. The motivation for the search for <n,d,k,t> graphs was that tasks could be assigned to clusters in an arbitrary manner, hence the need for graphs with low diameter. However, it should be possible to take advantage of the structure inherent in task communication to determine optimal assignments of tasks to clusters. The problem is as follows. Assume that the computations to be carried out are expressed as graphs, the nodes of which are tasks and the edges indicate communication between tasks. For a set of such computations, find an optimal embedding onto the underlying network graph. It is not obvious just what constitutes optimality, but minimizing the longest communication path seems to be a good choice. It is noted that this problem is related to the VLSI placement problem, although our problem does not have the rectilinear structure of the VLSI problem.

- Effects of Combinations of Failures. Our design effort so far has assumed that faults are handled shortly after their occurrence. We have avoided considering the recovery from multiple failures, e.g., a massive transient at the same time as a link failure. Some effort should be given to this more general case.

Chapter 2

The Analysis of Transient Faults

Introduction

Faults in computer systems are of two kinds, solid and transient. A solid fault is one in which some component of the system fails and will continue to fail for all subsequent uses. A transient fault is one in which some component of the system is temporarily deranged and fails in use, but in which that component subsequently recovers, without repair action, and in subsequent use the component does not fail.

Transient faults may be caused by thermal noise in a marginal component, by cosmic rays or alpha rays, or by electromagnetic interference. For typical transient faults, the faulty component is deranged for only microseconds or at most milliseconds, though the errors resulting from the fault may persist for much longer. It is difficult to obtain dependable information on the frequency of transient faults under operational conditions, because current systems are not instrumented to distinguish between solid and transient faults. However such information as is available indicated that transient faults will occur more frequently than solid faults, at perhaps ten times the rate.

It is important to distinguish between solid and transient faults, since processors suffering from a solid fault are removed from the system configuration. In contrast, processors subject to a transient fault are permitted to remain in the configuration. In other designs, the initial action taken for both solid and transient faults is the same - the processor is temporarily removed from the configuration pending tests to determine whether it is working and can be readmitted. This approach is not used in SIFT because:

- Processors diagnosed as having a solid fault are never readmitted to the configuration after they have been removed, even if the off-line diagnostic tests cannot detect any fault. The coverage of the diagnostic tests is not high enough to ensure that the benefit from readmitting good processors to the configuration outweighs the loss in reliability from readmitting defective processors.

- Processors diagnosed as suffering from a transient fault are not removed from the configuration, even temporarily, since the short duration of transient faults ensures that the actual faulty condition will not last even as long as the time required for error recognition and reconfiguration.

The reliability modelling results of the SIFT project [1] analysed the ability to distinguish between solid and transient faults. Assuming that transient faults are substantially more frequent than solid faults, it is important for the error diagnosis of the system to be able to recognize transient faults.

- If transient faults are incorrectly diagnosed as solid, resulting in working processors being deleted from the system configuration, the rate of system failure due to exhaustion of spares is greatly increased.

- If a solid fault is incorrectly diagnosed as a transient, the effects on system reliability are much less deleterious. The solid fault will generate further errors and provide further opportunities for repeating the diagnosis and recognizing the solid nature of the fault. The system is at risk to the occurrence of a second fault, whether transient or solid, during the time interval before the fault is correctly diagnosed.

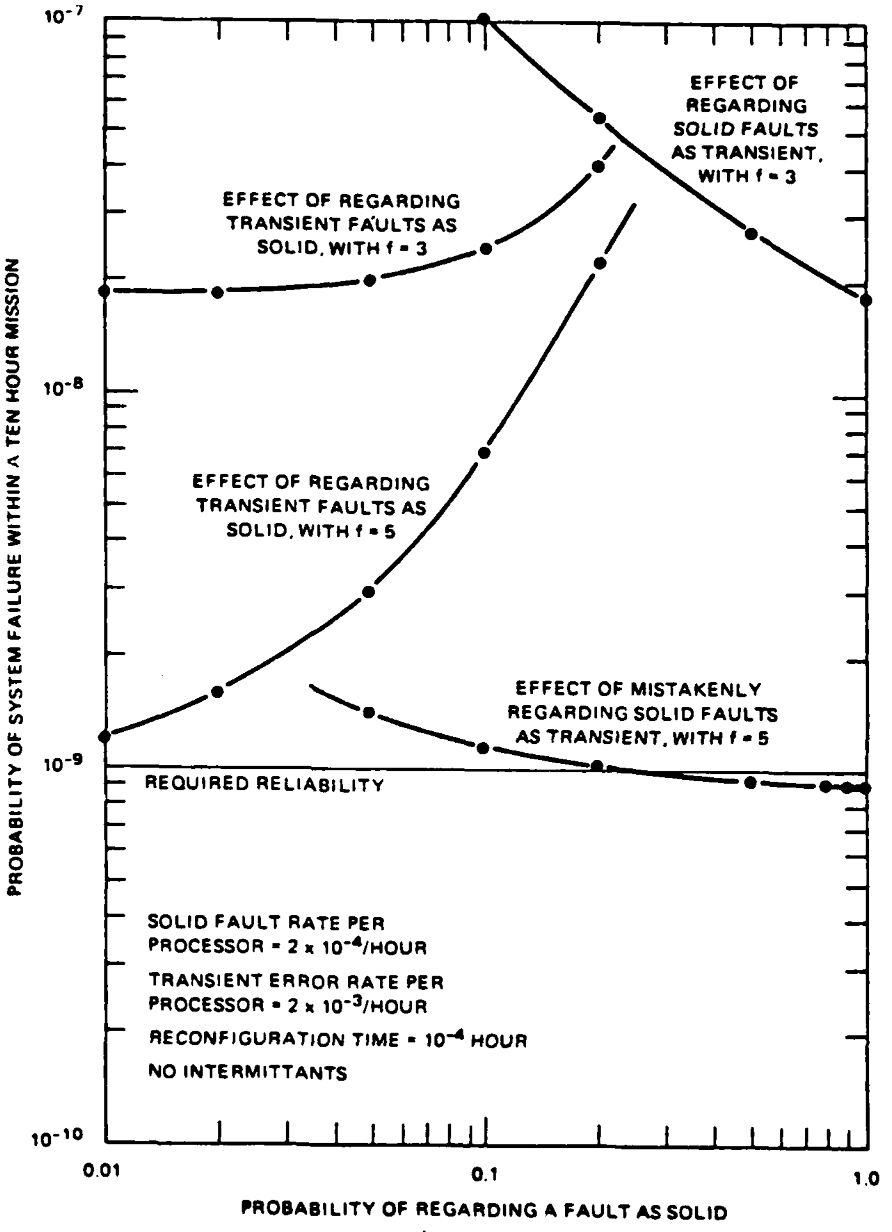


Figure 15. The Effect of Discrimination between Solid and Transient Faults

Figure 15 contains results obtained from the reliability model for SIFT, showing the probability of system failure within a 10 hour flight*. It is evident, particularly where critical functions are protected by five-fold voting (f=5), that a relatively small probability of regarding a transient fault as solid has a much bigger effect on the probability of system failure than does the corresponding probability of regarding a solid fault as transient. But, of course, it is essential to recognize solid faults; regarding all solid faults as transient is devastating to the reliability of the system.

Consequently, the ability of the system's error diagnosis routines to distinguish between solid and transient faults is very important.

2.1 Solid Fault Types

Many solid faults are catastrophic and either prevent the computer from generating any results at all or cause almost all results generated to be erroneous. However some faults, though solid, produce erroneous results only in rather specific circumstances. Such faults generate periodic errors, produced more or less frequently whenever those specific circumstances occur. A solid fault that generates errors only infrequently can be difficult to distinguish from a succession of transient faults.

Some faults will never yield an erroneous result, for the particular components are never actually used to produce the results in question. Other faults yield errors, not on every execution of the task, but only for specific data values, resulting in errors every few milliseconds, or seconds, or minutes. Experiments have been performed at NASA Langley Research Center to investigate the proportion of solid faults that do not generate immediate errors in the results of ap-

*It is important to note that these results are based on plausible but arbitrary component failure rates. Consequently the results can only be of qualitative significance. Quantitative measures of the reliability must be derived from careful measurement of actual component failure rates under operational conditions.

plication programs. Unfortunately these experiments have, for obvious economic reasons, considered only the proportion of faults that result, or do not result, in an error within a relatively short time interval. It is important to extend this work to determine the shape of the tail of the error-generation-frequency distribution, and it is to be hoped that the team at NASA Langley might consider such an experiment for AIRLAB.

During the period between the time when a solid fault occurs and the time when the fault causes an error, the fault is "latent". Latent faults are of course undetectable. The duration of latency of the fault is not significant, and latent faults are no more damaging to system reliability than simple faults, provided that the fault is "uncorrelated". While the latent fault remains undetectable so long as it is latent, it can also do no damage so long as it is latent. Only when the fault generates an error is there any risk to the system, and the duration of the previous period of latency is of no significance, provided that the error is generated at a random moment in time.

Correlated latent errors present a significant risk to the reliability of the system. A correlated latent error remains latent until some other error also occurs, and thus is manifested only in a double error situation. There are two ways in which this can occur:

- The latent fault can be such that the only circumstances in which errors are generated are those in which other errors are already present. Such a fault might damage only the operation of the error detection, diagnosis, or reconfiguration.

- The latent fault can be such that errors are generated only during a specific infrequently performed, but critical, function (for instance the autolanding functions). There is a risk that two processors might each be affected by such a latent fault, undetectable until the function is invoked and then yielding a double error.

This analysis, and indeed the whole SIFT design, does not address correlated faults.

There can also exist faults that are solid in that their defect is due to a physical cause that is permanent, but which generate errors only infrequently due to some physical aspect of the nature of the fault, rather than due to the nature of the processing being performed. Such faults are referred to as "intermittent" and are sometimes caused by cracks in conductors or by loose particles in packages. An intermittent fault resembles a succession of transient faults. The duration of each error generating event of an intermittent fault is usually rather longer than for a transient fault, and the frequency of such event is usually much greater than the frequency of transient faults in a properly working processor. The reliability analysis for SIFT indicated that the transient fault analysis algorithms are well able to protect the system against intermittent faults.

2.2 Error Generation and Detection

Faults, whether solid or transient, are manifested only through the errors that they generate, whether those errors are in the results of the application tasks or errors in the results of a diagnostic test sequence. This immediate error is of course only an incorrect result. To act on the error requires that it be detected, that a checking mechanism be capable of recognizing that the result is indeed incorrect and thus that an error, and by implication a fault, exists. Once the error is detected, it must be diagnosed that a specific type of fault is the cause of the error, and that some recovery or reconfiguration action is appropriate.

In typical low reliability systems, error detection is very poor and the degree of confidence that any particular erroneous result will be noticed is low. High reliability systems, such as SIFT, in contrast have very good error detection and almost any error will be detected.

Even though a fault may cause errors to be generated 'immediately', the errors are not detected, and thus the existence of the fault is not recognized, until the erroneous results are subjected to the voting or other error detection checks. The results of high priority tasks are needed for use by other tasks within a short period of time, and thus must be voted or checked very promptly, certainly within a few milliseconds. Thus, a fault that causes errors in the results of high priority tasks can be detected soon after the fault occurs.

However, many systems contain background tasks whose results are not needed immediately. The execution of such tasks may be spread over several seconds, or even longer, and the results may not be voted until some convenient moment long after they were generated. A fault that yields and error in the results of a background task may not be detected until seconds, or even minutes, after the fault occurs.

This has two effects:

- During the interval between the generation of the erroneous result for the background task and the masking of that error by voting, the system is vulnerable to the occurrence of a second fault. Fortunately, background tasks are usually not very critical and a rather higher risk of failure of such a task can be accepted. Results that are very critical must be voted at frequent intervals to ensure that errors are masked promptly, thus reducing the risk of error accumulation between masking. This frequent voting of critical results is necessary even if processing of those results is required only infrequently.
- A single transient error may occur and damage the results of several tasks. The erroneous results of high priority tasks will be detected quickly, but further error reports will continue to be generated for some time as other results of lower priority are voted. This might confuse the error diagnosis routines into thinking that the error that has occurred is persisting in generating errors, and thus may be solid. It might also confuse the error diagnosis routines into thinking that multiple faults had occurred.

It might be hoped that the nature or appearance of the error detected might provide an indication as to the location and type of the fault that caused it. Unfortunately, the errors detected are often of the form of an incorrect result and it is difficult to ascribe a cause from such meager information. Further, a "malicious" fault may masquerade as some different type of fault. It is essential that such deception should not permit the successive removal from the configuration of working equipment until system failure results.

Any one error report originates at a single point in the system and the report must be replicated for analysis by the necessarily replicated global executive routines. As for any other information that originates at a single point, interactive consistency or interactive convergence techniques must be used to ensure that the replications are consistent. Even when a component reports itself to be faulty, it is essential to use interactive consistency techniques to detect that a processor has reported itself faulty to one neighbour and not to others, and situation indistinguishable from that in which the neighbour falsely claims that the processor has reported itself faulty.

2.3 The Analysis of Error Reports

An error report is certain information that an fault has occurred, but less certain as to what fault and when. If interactive consistency techniques are used, a report by processor A of an error in the results of processor B for iteration i of task k provides the information that:

- the fault existed in either processor A, or processor B, or the link between them
- the fault existed at some time since the start of the data-window for iteration i of task k

The global executive routines must make use of the combination of many error reports to deduce the true nature of the underlying fault. The basic algorithms used in SIFT are described in [1]. We discuss here three aspects of fault diagnosis:

- identification of, and action on, link failure,
- identification of transient faults,
- identification of low error rate solid faults.

2.3.1 Identification of Link Failure

When a processor fails, it will probably generate erroneous results and broadcast them to all of the other processors, resulting in a large number of error reports from which it is easy to diagnose which processor has failed.

Less probably, a processor might suffer from a fault that causes it to generate erroneous error reports even though the results being voted were correct. If that processor is detected by the global executive to be generating many error reports, claiming errors in several other processors, all unsupported by reports from other processors, the diagnosis is again relatively easy.

But failure of the link between two processors results in error reports in which one processor systematically reports errors in the results of just one other processor, without any corroboration from other processors. The exact location of the fault may be:

- in the physical link itself,
- in the transmitting circuitry of the broadcasting processor, after the point at which the common broadcast signal has fanned out into separate signals for each destination,
- in the receiving circuitry, or the result buffering, or the voting software, or the error reporting software, of the processor reporting the error.

Because continued operation of SIFT requires full connectivity between all processors of the configuration, and because continued operation with a faulty link exposes the system to failure should another fault occur, it is essential to reconfigure the system to a reduced configuration in which the faulty link is not required, i.e. to a configuration without one or other of the two processors at either end of the link.

If the fault is simple, it matters little which of the two processors is to be reconfigured out of the system. But it is very important that a malicious fault should not be able to exploit the choice to remove systematically a succession of other processors.

The algorithm recommended is:

following a link failure event in which processor B reports errors in the results of processor A, without corroboration,

- if processor A is not on probation then:
    - processor B is removed from the configuration,
    - processor A is recorded as being on probation,
- if processor A is on probation then processor A is removed from the configuration.

The choice is made to favor removing processor B, rather than processor A, from the configuration because there is very little logic in processor A after the fanout point at which the common broadcast signal is split into separate signals for each destination. Consequently is is relatively improbable, though not impossible, for a malicious fault to develop in that small amount of logic within processor A. In contrast, the amount of logic, both hardware and software, in processor B that is capable of producing the symptoms is quite large.

However, the algorithm must guard against the possibility of a malicious fault in the small amount of logic in processor A. Consequently, processor A is placed on probation. Thus, if a malicious fault in processor A should succeed in causing processor B to be removed from the configuration, any subsequent attempt by processor A to repeat the attempt, say on processor C, results in the removal of A rather than C. Consequently, the rather improbable malicious fault in processor A can cause two processors to be lost, but no more than two.

2.3.2 Identification of Transient Faults

The identification of transient faults is based on their short duration. It is assumed that a fault is solid if it persists, i.e. continues to generate errors, for more than some period of time (known as the ***solid/transient discrimination*** *interval*). Typically that period of time might be set to say 200ms or 300ms, for almost all transient events are much shorter. Faults that generate only single isolated errors, or short bursts of errors, are assumed to be transient.

In systems in which all tasks operate at iteration rates shorter than the solid/transient discrimination interval, such as SIFT Mk I, it is relatively easy to distinguish solid from transient faults. Such system detect all errors within one iteration, and any fault that causes errors to be detected in two or more iterations can be assumed to be solid.

Future systems will contain tasks that operate at very different iteration rates, and some of those iteration rates will be much longer than the solid/transient discrimination interval. The detection of errors in the results of slowly iterating tasks may be delayed for a period comparable to the iteration interval of the task. For certain navigation and fuel management tasks, this delay may be many seconds or even minutes. Thus, even for a transient fault of short duration, if the results of lower priority tasks have been affected then error reports may be generated periodically over a relatively long interval of time. Consequently it does not suffice to assume that a solid fault is indicated by error reports spread over an interval longer than the solid/transient discrimination interval.

The proposed algorithm is based on the concept of *fault windows*, the interval of time somewhere within which a fault must have existed to cause the observed error symptoms. We will consider two types of fault windows:

- error report windows,
- fault event windows.

An error report window is the interval within which a fault must have existed to result in the observed error report. The error report window, for an erroneous result from task A, extends from the earliest time of voting of any input value to task A until the completion of interactive consistency on the error report.

A fault event window is the interval within which a fault must have existed to cause several error reports, and thus is the intersection of the of the error report windows for each of the error reports.

The proposed algorithm for discrimination between solid and transient faults is:

- initially the fault event window is set to empty.
- when a fault report is received,
  - ▶ the error report window for that report is computed,
  - ▶ if the fault event window is empty then an new fault event window is created equal to the current error report window,
  - ▶ if the fault event window is not empty, but the intersection between the fault event window and the error report window is empty, then again a new fault even window is created equal to the current error report window,
  - ▶ if the intersection of the fault event window and the error report window is not empty, the fault event window is set to that intersection.
- if the end of the fault event window is so long ago that no current fault could generate an error report window to intersect it, the fault event window can be set to empty.
- every time that a new fault event window is created, analysis is made of the frequency of fault event to determine whether reconfiguration action is required.

The behavior of this algorithm is illustrated in Figures 16 - 19. In Figure 16, the fault event window is initially empty. Thus the error report window is computed and a new fault event window is created and set equal to the error report window.

In Figure 17, a further error report has been received whose window overlaps the fault event window. We assume that the same fault generated both error reports. Thus the fault event window is reduced in size to the intersection of the two windows. In Figure 18, it is still possible that the same fault caused all three errors and thus the fault event window is again reduced in size.

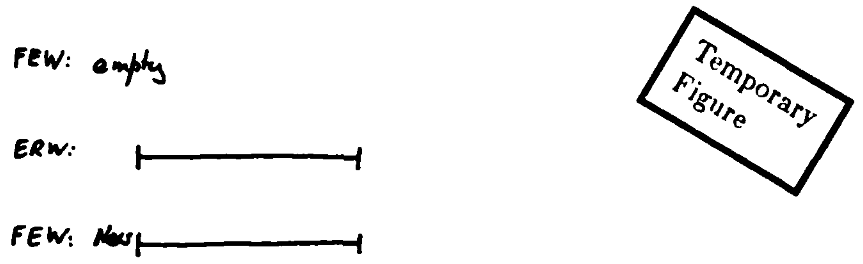


Figure 16. An Error is Reported when the Fault Event Window is Empty

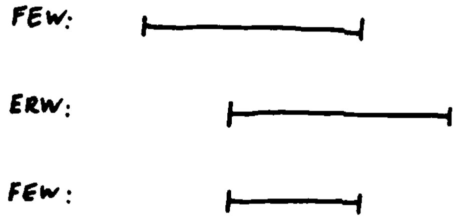


Figure 17. An Error whose Window overlaps the Fault Event Window

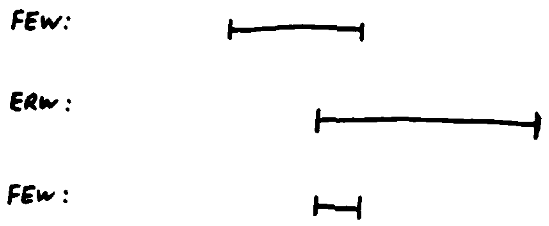


Figure 18. A further Error whose Window overlaps the Fault Event Window

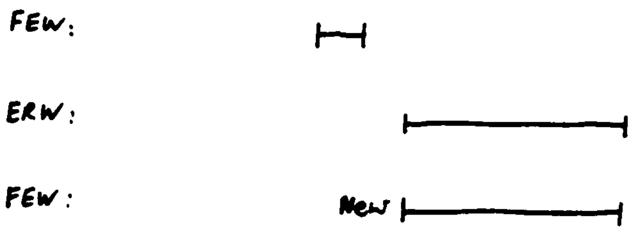


Figure 19. An Error whose Window does not Intersect the Fault Event Window

But in Figure 19, the next error report window no longer intersects the fault event window, and it is not possible for a single transient fault of short duration to have caused all of the errors that have been reported. Thus we create a new fault event window, equal to the error report window, and start the analysis to determine whether the frequency of faults requires reconfiguration (see the section below).

2.3.3 Identification of Low Error Rate Solid Faults

The discrimination between solid and transient faults depends on the observation that a transient fault is of short duration, and thus on the assumption that a set of errors, generated in some short interval and not followed by other errors, have probably been generated by a transient fault. But some solid faults are such that only occasional results are damaged by their presence. The equipment is definitely broken, but the nature of the fault is such that many correct results can be generated and only a few are erroneous. Unfortunately, it is unlikely that in service it will be possible to diagnose faults sufficiently to distinguish solid faults that generate errors only occasionally from transient faults. Consequently we do not distinguish between transient faults and low rate solid faults, but rather aim to determine whether the rate of occurrence of such faults is such as to damage the overall system's reliability.

If a processor suffers from a solid fault (or a transient fault) generates errors only occasionally, we must consider whether retaining that processor in the configuration improves the reliability of the system or reduces it. Occasional erroneous results from the processor can be masked by the voting algorithms, but retaining the processor in the configuration increases the risk that its erroneous result will coincide with some other erroneous result, causing system failure. Removing the processor from the configuration eliminates the risk of coincident errors, but increases the risk of exhaustion of spares should several other processors fail.

A preliminary analysis of this problem was performed using the reliability model for SIFT. It is important to note that the results here are only indicative, and that the modelling should be repeated with more accurate data. The model was used to compare the reliability of two SIFT configurations:

- A five processor SIFT, with four normal processors and one processor set to generate occasional (transient) errors,
- A four processor SIFT, with all normal processors.

A normal SIFT processor was assumed to have a solid fault rate of 2×10^{-4}/hour and a transient fault rate of 2×10^{-3}/hour. The error rate of the 'special' processor was varied to investigate the effects of a higher than normal transient event rate.

For a SIFT system in which only three way voting is performed on critical functions, it was found that the 4 processor system became more reliable if the transient rate of the 'special' processor exceeded 10^{-1}/hour. In effect, a processor that suffers even a single fault per fight damages the reliability of the system, a rate that is only slightly greater than the expected transient rate for normal SIFT processors.

For a SIFT system in which critical functions are five way voted, it was found that the 5 processor system, containing the 'special' processor, remained more reliable even for special transient rates as high as one per few seconds. When ample error masking was available, the risk of coincident errors was not significant.

This investigation should be extended. For instance, no investigation was made of the effect of remaining mission duration of the discrimination, nor was consideration given to situations in which more than one processor has a high transient fault rate. The analysis should also be performed for systems with initial numbers of processors other than 5.

Chapter 3

Application of Ada to Fault-Tolerant Systems

3.1 Scenario and Goals

SIFT successfully demonstrated reliable computation for aircraft flight control applications through replication of flight control programs on multiple computers. Fault masking was achieved by broadcasting results of replicated tasks and majority voting. In SIFT, task replications, executing on distinct processors, maintain loose synchronization using a preplanned schedule.

The characteristics of the SIFT software structure are:

- A fixed set of user tasks
- Tasks are periodic and executed with fixed frequency
- Communication between tasks is limited to a fixed number of "results", broadcast at the end of each task iteration. Tasks share no storage and have no communication during execution.
- Communication between successive iterations of a task is limited to the same broadcast "results". No storage is preserved between task iterations.

- A preplanned schedule ensures that task results are available when required by other tasks; tasks are never required to wait for input values.
- Errors in broadcast results, caused either by processor or communication faults, are masked by majority voting. Since broadcast results are the only information representing task state which is retained, no other form of fault masking is required.
- No form of masking of errors due to incorrect programs is included.
- Errors detected during voting result in reconfiguration.
- Discrimination is provided between solid and transient faults. Transient faults are masked by voting and do not cause reconfiguration.
- Reconfiguration consists of choice of a new, preplanned, schedule and allocation of tasks to processors.
- Based on individual reliability requirements, tasks can be selectively replicated to any necessary degree. These replications can be allocated to processors to balance processor load.

The orientation of the SIFT design is towards predictability and reliability rather than flexibility. An advantage of the approach is the very simple, and therefore inherently more reliable, nature of the Executive software. The simplicity of the approach used in the Executive software imposed many constraints on the user and exposed aspects of scheduling, communication and replication. These constraints are not inherent in the "SIFT concept" – they were imposed to allow a very simple implementation.

In this report, we investigate to what extent the concept of SIFT can support a more general user interface. We consider a system in which the structure of the user program is not constrained by the needs of fault tolerance, and in which the system is not as dependent on the user to specify management of information and resources in the system. In particular, we seek to permit a more dynamic program structure.

To this end, we consider the Ada virtual machine and investigate building a *reliable* Ada machine. Our goals are:

- To provide fault-tolerant support for a wide class of Ada programs.
- Ada programs should be unchanged, except for advisory directives.
- To allow greater asynchrony between executions of Ada program replications.

3.2 Considerations in Providing a Reliable Ada Machine

In investigating a reliable Ada machine, we continue the SIFT approach of replication on independent processors, with error detection and masking based on majority voting. For majority voting to suffice to detect and mask errors, *all replications* of the program executing on working processors are required to produce *exactly the same results.* Even non-deterministic programs must adhere to this requirement – all instances of the program must behave identically.

3.2.1 Effect of Non-determinism

Ada provides a tasking facility, which inevitably introduces non-determinism. This non-determinism results from direct interaction between tasks and from access by tasks to shared global variables. Ada restricts access by tasks to shared global variables, so that all non-determinism in Ada programs stems from direct communication between tasks. The Ada mechanism for interaction between tasks is the *rendezvous.* The rendezvous involves a task which calls a rendezvous *entry* and a task which *accepts* the entry call. An entry call, equivalent to a procedure call (with parameters), suspends the calling task until the entry is accepted and a rendezvous is completed. When the called task accepts the call, a rendezvous occurs, and the body of the accept procedure is executed. Following completion of the accept procedure, both tasks are allowed to continue asynchronously. If a task reaches the accept point, it is suspended until called.

Non-determinism is introduced by several tasks contending asynchronously for the same accept procedure. Ada does not provide any guarantee of timing for the concurrently executing tasks, and thus does not determine which task will reach the call first. This timing can be influenced by lower level factors such as system scheduling, interrupt handling, etc. These factors may vary from processor to processor. In order for our majority vote masking to succeed, we must be able to guarantee that all processors executing the multiprocess Ada algorithm *accept the same entry call* into the rendezvous. We shall refer to this as a *consistent rendezvous.*

Tasks in Ada are *objects*, which may be dynamically created and dynamically terminated. One does not have any predefined configuration of tasks. It is possible for an Ada program to contain an arbitrary number of instances of an Ada task type.

3.2.2 Periodic Voting

The execution of an Ada program, of course, can be of arbitrary duration. Reliability requirements demand periodic voting to detect and mask errors. When should these votes be performed? Each vote in each processor must be performed at exactly the same point in the computation. The moment of voting cannot be determined solely on the basis of time, since different processors may be in different states at that time. It also is not possible to determine for an arbitrary program how to embed vote requests in the program to obtain votes with appropriate periodicity. For an arbitrary program there will be no obvious iterative structure which could guide this choice. One must also take into account that some points in the program may be more appropriate for voting because of cost or effectiveness.

A second issue concerns what information need be voted. Sufficient data must be voted to detect that tasks instances executing on different processors are performing the same computation. In SIFT, because voting is performed on task results at a time when the task has terminated, only those results need be voted.

For a more general Ada program at which votes are taken periodically, there is no explicit indication of what constitutes "results".

3.2.3 Resource Management and Scheduling

Many potential applications of fault-tolerant computing involve real-time performance constraints. In SIFT, these constraints are guaranteed to be satisfied by a rigid, preplanned schedule. An Ada-program, designed to meet the same constraints, depends on dynamic interaction between the program and the Ada scheduler. We expect that the Ada programs to be rendered fault tolerant will already contain the resource and scheduling strategies necessary to meet the real-time constraints. The introduction of fault tolerance should not perturb this basic strategy, although there will inevitably be some overhead introduced as a result of the additional mechanism.

3.2.4 Where to Embed Fault-Tolerance Mechanism

The additional mechanism needed to achieve fault-tolerance can potentially be introduced at one of three levels:

- Below the level of the Ada run-time system.
- Within the Ada translator and its run-time support.
- Above the level of the Ada virtual machine.

The first alternative, implementing fault-tolerance below the level of Ada, implies the Ada translator and its run-time support can be completely unchanged. In order to accomplish this, one would have to introduce a fault-tolerant version of the *processor architecture* assumed by the Ada translator. This approach is certainly feasible, using mechanisms such as dual-dual. The problems to be solved turn out to be comparable to those using the other alternatives, but the mechanisms cannot exploit the structure of the Ada program to reduce the cost of the additional reliability.

The third alternative would consist of an Ada package, programmed entirely in Ada, to implement the necessary mechanisms. This package would reproduce to the Ada user program a fault-tolerant virtual machine equivalent to the original machine. This would allow a highly portable solution, allowing the fault-tolerance mechanisms to be applied to any system supporting the Ada. To accomplish this, all necessary mechanisms would have to be expressible *within Ada*. Thus, the consistent rendezvous must be programmed using the rendezvous facility for communication between processes. Even with the aid of a preprocessor to introduce additional statements into the Ada program, it would still be necessary to augment the Ada compiler and run-time system to provide information not normally accessible to the Ada program.

The second alternative, that of modifying the compiler and run-time support of Ada, permits more efficient implementation of the necessary fault-tolerance mechanisms. This is at the expense of requiring changes to a rather complex compiler and run-time system, and results in a translator-specific implementation of fault-tolerance.

In the following sections, we explore the capabilities necessary to support the fault-tolerance techniques, and comment on the difficulties in implementing fault tolerance within the Ada system.

3.3 Mechanizing Fault-Tolerance for Ada

In this section, we describe mechanisms to support a fault-tolerant Ada virtual machine. It requires both error detection and masking support and a mechanism to ensure consistent rendezvous. In presenting the major ideas, we treat Ada programs, possibly itself implementing a multiprocess algorithm, as a monolithic program, replicated in its entirety.

3.3.1 Error Detection

The basic approach will be based on replication of the Ada program on independent processors. We use a majority voting scheme to detect faults in a minority of processors - the consensus in such a configuration is assumed to be correct.

In SIFT, each processor uses only *voted* values as inputs - achieving immediate error masking. Here, where there is no notion of distinct inputs, we have no concept of masking input values. Rather, we use majority voting to *detect* errors. Each processor uses only local state information in performing its computation. Following an error, a processor will continue to compute erroneously, but cannot influence other processors' computations. The error will be detected during majority voting, leading to later fault diagnosis and reconfiguration. Error masking occurs as a result of reconfiguration to exclude dependence on erroneous processors.

To detect any erroneous computation, it is necessary to vote the *entire* state of the program. Voting any less than the entire state could permit an undetected error that might adversely affect the future computation. The program state, of course, can be rather extensive - consisting of the run-time stack, heap, expression stack, and any other run-time management information. Rather than broadcast and vote this potentially large amount of data, we compute a *signature* of the state. This signature should be an encoding of the state with sufficiently high probability that distinct states map to distinct signatures. Furthermore, signature calculation should not be highly correlated with the computation; if by chance an erroneous chance reduces to the same signature value as that of the consensus, further computation and a further vote should have a low probability that equal signatures will again occur. Digital techniques such as [6] satisfy these criteria. The length of the signature can be adjusted to meet the required probability of immediate error detection.

One consequence of a multiprocess Ada program is that different instances of the same program may contain tasks scheduled differently on different processors.

Programs may *never* be in exactly the same state. Consequently, we cannot in general vote entire Ada programs. Rather, voting must be done on a task-specific basis. Therefore, it must be possible to determine a partitioning of program state into task states. This raises several problems.

First, there may be no simple way to partition global state into task states. Secondly, even when such a partitioning is possible, it may be impractical to deduce. Because processes may interact through global variables, the appropriate partitioning may be only dynamically determinable. In order to provide a practical solution, we will disallow reference to global variables, forcing all task communication to be via rendezvous calls.

For any partitioning, it is necessary that the combined partitions account for the entire global state of the program. Votes at different times on task states must ensure that the net effect is to guarantee that no information will escape being voted. To ensure this, it is necessary not only to vote the information inside the task state, but all information being communicated between tasks. Since all information flow occurs by rendezvous, it is sufficient to additionally vote all values passed as parameters by entry calls. Consistent with our global variable restriction, no **in out** parameters or **access** values may be passed as entry parameters.

Assuming the implementation of Ada is such that each task has a local run-time stack and expression stack, voting the current state of a task necessarily requires voting the value of these stacks. This cannot be done, of course, above the Ada virtual machine, but can be accomplished by functions added to the Ada run-time support. It is not a safe assumption that heap space is partitioned in a similar manner. Any use of the heap, to allocate access variables, for example, must be traceable to a single task. Voting must include all stack and heap values.

Vote values are generated in one of two ways. Upon encountering a user-supplied vote *pragma*, a signature of the task's entire state is computed and broadcast to the other processors. Upon entry call to a rendezvous, signatures of input **in** parameters are computed. Upon return from a rendezvous, signatures for

out parameters are computed. Each signature is tagged with a task identification and a sequence number which uniquely identifies that vote.

Having established that state signatures are broadcast to all processors, we now describe two possible algorithms for detecting and reporting errors. Because no processor is dependent on values computed by other processors, there is no need for synchronization between processors at vote points. However, some synchronization points are necessary in order to avoid an unbounded amount of storage necessary to hold signature values until a consensus is possible.

The simplest algorithm provides storage for one signature per task instance. Processors can proceed asynchronously up to a vote point. No task instance can progress beyond a vote point until all other instances have reached the previous vote point. To ensure that a minority of failing processors cannot indefinitely delay a vote, we must include a timeout mechanism in this vote. Timing starts when a majority of processors have submitted values. It is assumed that it is possible to establish an appropriate timeout value for each task and that the scheduling can maintain the skew between instances of the task on working processors to less than this value. At the expense of increased storage, it is easy to extend this algorithm by storing additional signatures, thereby allowing greater asynchrony.

There is an alternative algorithm that allows much greater asynchrony without storage penalty. Since voting is used for error detection rather than masking, it is not necessary to vote every signature value separately. Rather, we aim to maximize the number of pairwise comparisons between values generated by different processors.

The voter stores, for each pair of task instances, one signature, its tag, and the id of the processor that generated it.

- If no value is currently stored, a signature triple arriving from either processor can be stored.
- If a signature triple arrives from same processor as that of the outstanding triple,that signature is ignored.

- If the arriving signature is from the other processor:
  - it is ignored if its sequence number is smaller than that outstanding;
  - it is stored if its sequence number is greater than that outstanding;
  - it is voted if the sequence number of the arriving triple is equal to the outstanding triple. Error reports are generated and broadcast when discrepancies in the vote are encountered.

To implement this scheme, we require that successive signature values be computed cumulatively, i.e., that the previous signature be included in the calculation of the next. Thus, each vote includes all previously computed signatures, and errors can be detected even though every signature value is not voted independently. As for the simple algorithm, a timeout mechanism must be used to ensure detection of processors that generate no signature or infrequent signatures. Processors that generate signatures with inappropriate sequence numbers will also be detected.

The maximum interval between votes in this scheme is equal to the maximum interval between generation of signatures plus the maximum skew between the execution of the task on different processors.

3.3.2 Error Masking

The voting of task state, as described above, can only be used to detect errors. Reliable operation requires also that error masking be provided:

- to mask transient errors,
- to move task instances from processors deemed faulty to other processors.

The algorithms by which the SIFT Global Executive diagnoses faults from the error reports, and distinguishes solid from transient faults, are equally applicable here and need not be described. In SIFT, voting automatically masks transient errors and the Global Executive need take no action. When the Global Executive diagnoses a solid fault, tasks must be assigned to execute on other processors. Those processors have already obtained the required input values, with errors masked by prior voting, and can immediately assume the tasks. For the current scheme, however, the Global Executive must issue explicit directives to mask both solid and transient faults. This masking must be performed by copying ***the entire program state*** from a processor deemed to working correctly. The entire state can be copied at once, though it may be possible to copy on a task by task basis, thus reducing the time for which processing is suspended.

Chapter 4

Asynchronous Voting

The need for reliable computation has induced many designs for fault tolerant computer systems based on the replication of the processors and appropriate error detection and masking algorithms. Typical of such systems are SIFT and FTMP, which use majority voting for error masking, and Stratus, which uses a dual-dual structure for error masking. It is clear that these approaches, coupled with the steadily improving reliability of components, now allow the construction of very reliable systems.

All fault tolerant systems depend on some form of error masking algorithm, coupled with error detection to allow the repair of faults. Some such systems depend on backward error correction, in which a result is computed, the acceptability of that result is checked, and in the event of error the computation of the result is repeated. Typical of such systems are classical Checkpoint-Restart systems and Recovery Blocks. Backward error correcting algorithms necessarily incur a significant overhead for repeating the computation when an error is detected, and also involve an acceptance test on the results, a test that is usually system and application specific. We do not consider backward error correcting systems in this paper but rather we examine Forward Error Correcting systems, in which the results are computed in a redundant form that allows error masking without repeating any computation.

Two forward error correcting algorithms are currently used for masking processor errors in reliable systems, majority voting and dual-dual. The majority voting approach can mask errors caused by one faulty channel out of three, while a dual-dual approach masks one faulty channel out of four. Both approaches have the advantage that they are completely application independent. However majority voting and dual-dual both depend for their operation on exact match comparison between results of computations. Thus, for successful masking of errors, it is essential that the fault free channels should generate identical results. Both algorithms guarantee, with only a single faulty channel and with fault free channels producing identical results, that fault free channels remain error free and continue to generate identical results.

Two questions arise from this. The first concerns whether there are any single point faults that could cause fault free channels to generate different results, thus invalidating the presumptions of both majority voting and dual-dual. We describe below a class of such faults and give algorithms for precluding them. The second question relates to the possible increase in the risk of common mode faults resulting from the need for all channels to perform exactly the same computation on identical data at approximately the same time. We show below that error masking algorithms can be devised that allow each channel to perform a different computation on different data at different times.

4.1 Loss of Consistency

Figure 20 shows a majority voted three channel system, with one faulty and two working channels. The successive levels of the diagram might represent distinct units within the channel, but equally they can represent successive iterations of a computation performed by the same units. It is clear that, provided that the two working channels generate identical results initially, each voting operation will receive as inputs two identical values and one erroneous value. The voters in the two working channels will therefore both produce the same value for the majority. Thus the working channels continue to generate identical results, and consistency between working channels is maintained. However, if at any time the three channels generate different results, the voters can find no majority and the system fails.

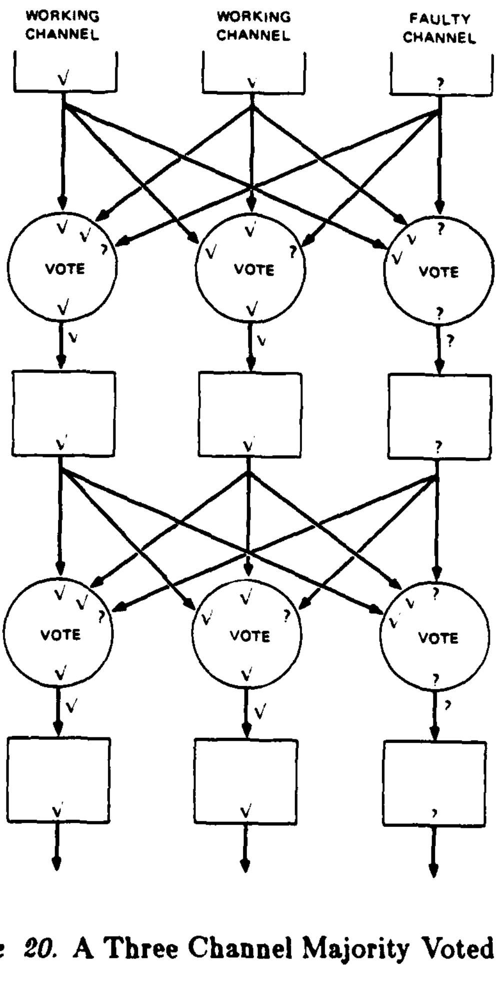


Figure 20. A Three Channel Majority Voted System

Consider Figure 21, which shows a system of three working channels and an input to that system from a single faulty source. The nature of the fault is that the source distributes different values to each of the three channels (the values A, B, and C). Even on a broadcast bus, such faults can result from marginal timing faults or from a marginal transmitter at the source and receivers with slightly different, but within specification, characteristics. More complex communication mechanisms, particularly where software is involved, permit many more such faults. The figure shows that, if the faulty source distributes different values to each channel, the three channels generate different results, the voters can find no majority, and the system fails.

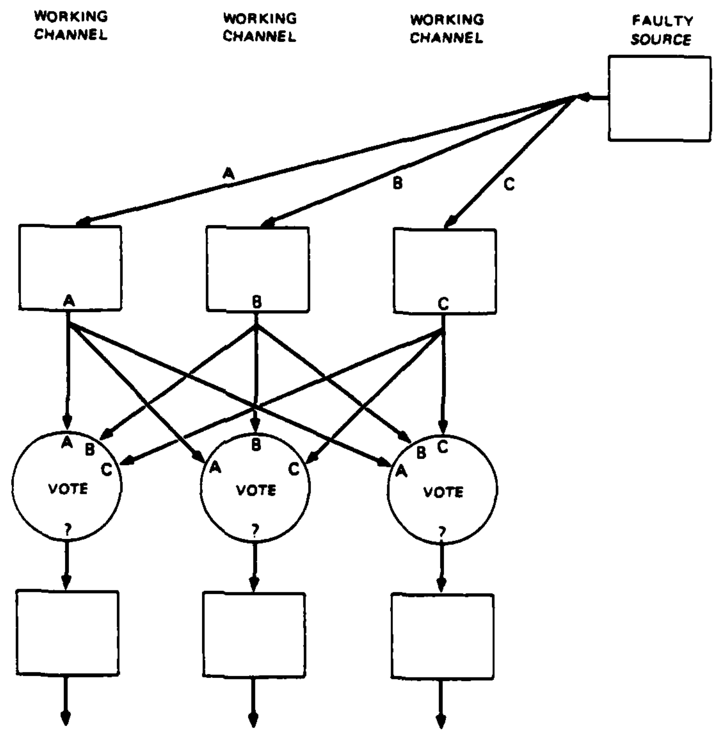


Figure 21. Distribution of Information from a Single Faulty Source to a Three Channel System

Figure 22 shows a three channel system with two working and one faulty channels. Here information present in just one of the channels is to be distributed to all three channels and be used in a replicated calculation. The faulty source distributes different values to the two working channels, and compounds the problem by repeating the same erroneous values (suitably transformed if necessary) in the next, voted, stage of the system. Note that not only do the two working channels continue to receive inconsistent values, even after voting, but also each of the two working channels can be mislead into believing that it is the other working channel that is faulty.

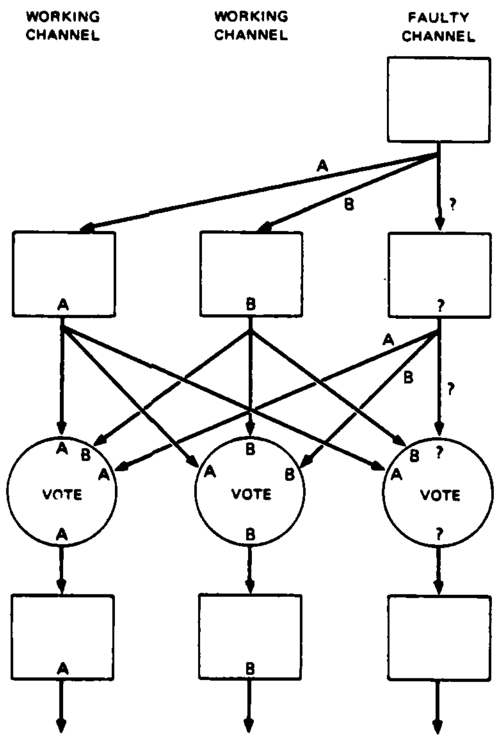


Figure 22. Distribution of Information from a Single Channel to Three Channels

The existence of this problem was discovered during the design of SIFT, a reliable aircraft control system, and is discussed in [4], where it is shown that no solution is possible in a purely three channel system. An algorithm, called the interactive consistency algorithm, is given for a four channel system containing a single faulty channel, and extended to the masking of N faults in a 3N+1 channel system.

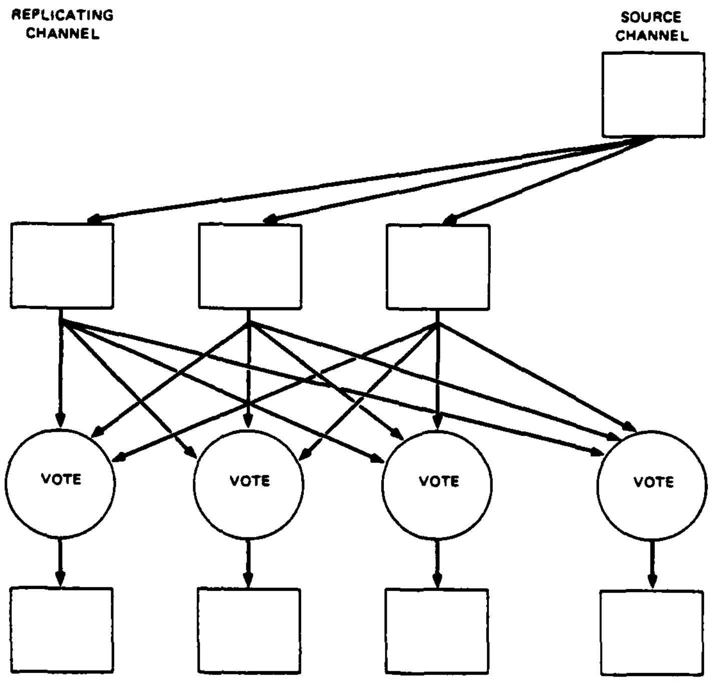


Figure 23. The Interactive Consistency Algorithm

The basic interactive consistency algorithm is given in Figure 23. One of the four channels is the single point source of the information, and the three other channels are used to replicate that information. Once the information is replicated, any or all of the channels can vote the replicated information with confidence that all voters in working channels will produce the same majority value, or alternatively all working voters will find no majority and will return a default value. For this algorithm to be effective against all faults, the channel that is the source of the information must be distinct from the three channels that perform the replication.

Consider the possibility that the source channel is faulty. It may then distribute different values to the other channels. The three replicating channels must all be working, and thus every working voter must get the same set of inputs. If at least two of the replicating channels have the same value, every working voter will find that value as its majority, while if all three replicating channels have different values, every working voter will return the default value. (If the source is faulty, the interactive consistency algorithm cannot of course guarantee a correct value from that source, but only a value that is consistent across all working channels.)

Consider the possibility that one of the three replicating channels is faulty. Now the source is necessarily working and will distribute the same correct value to each of the two working replicators, which will replicate it. Thus each working voter obtains at least two correct inputs and is able to produce the correct value as its result.

In SIFT, four circumstances were found in which a value from a single source had to be distributed to three replicated channels, namely:

- input from a sensor,
- error reports from a voter,
- interfaces between unreplicated and replicated tasks,
- synchronization of processor clocks.

The first three of these require the use of the interactive consistency algorithm to protect the system against malicious faults. The fourth is of special interest in that exact agreement is not necessary for clock synchronization, and thus slightly simpler algorithms guaranteeing approximate agreement suffice.

4.2 Maintenance of Approximate Consistency

In SIFT, as in many other fault tolerant systems, each processor has its own clock and operation of the system depends on these clocks remaining synchronized (to within 50ms in SIFT). Many prior systems used three channels, three clocks, and a clock synchronization algorithm based on each clock synchronizing itself periodically to the median clock of the three. It is instructive to consider why this "obviously sound" approach is invalid.

Figure 24 shows a system wi‹h two working clocks (A and B) and a faulty clock (C). We may assume that clock A runs slightly faster than clock B. Clock C presents to clock A an erroneous clock value indicating that clock C is running faster even than clock A, causing clock A to assume that it is the median clock. Thus clock A makes no correction to its value. Similarly, clock C presents to clock B a value indicating that it is behind even clock B, causing clock B to assume that it is the median clock and make no correction to its clock value. By this strategy, the faulty clock C can induce clocks A and B to operate without correcting their clock values as they gradually drift apart until the system fails. Single point component faults that could cause this "malicious" behavior have been found even in purely analog clock systems.

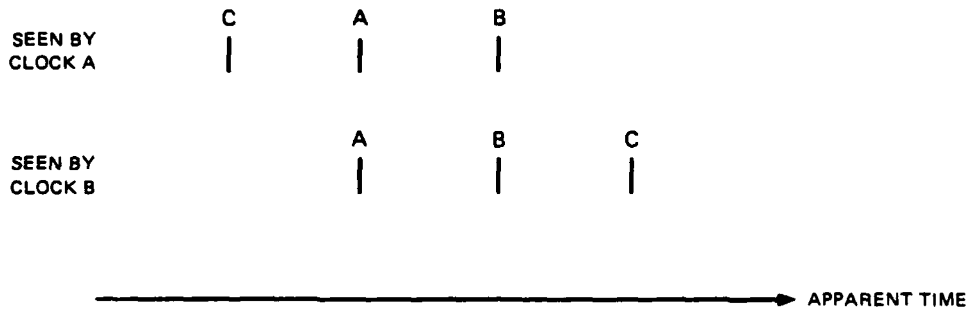


Figure 24. A Failure Mode of the Median Clock Synchronization Algorithm

It is tempting to attempt minor corrections to the three channel clock synchronization algorithms, aimed at preventing this behavior. As yet we have no rigorous mathematical proof that no three channel algorithm can exist, but we believe that the approximate agreement needed for clock synchronization requires the same number of channels as the exact agreement discussed above.

In SIFT, a *four channel clock synchronization* algorithm is used in which each clock is periodically resynchronized to the mean of the four clocks. To protect against wildly erroneous clock values, the algorithm imposes a bound within which a clock value must lie to be included in the averaging calculation. For n processors of which at most m are faulty, with R as the resynchronization interval and S as the time taken for resynchronization, and if ϵ is the maximum clock reading error and ρ the maximum rate of clock drift, it can be shown that the maximum skew between working clocks will not exceed

$$\frac{n}{(n-3m)}(2\epsilon + \rho(R + \frac{2(n-m)S}{n})).$$

A similar problem has been examined by L. Webster [7,8] in closed loop control systems. He found that use of a median voting algorithm in a three channel system favors the median channel, effectively disconnecting the two other channels from the closed loop. Without cross coupling between the integrators of the three channels, this results in uncontrolled accumulation of error terms in the integrators of two of the channels, rendering them useless for error masking. With cross coupling, the integrators are vulnerable to precisely the same problem as the clocks above.

The possibility of failure to maintain approximate consistency appears to exist in any three channel system containing embedded integrators.

4.3 Asynchronous Multichannel Systems

Existing fault tolerant multichannel systems using forward error correction, whether majority voted or dual-dual, depend on an exact equality between the result values of the various channels. To ensure this exact equality of their outputs, the various channels must all perform exactly the same calculation on exactly the same input values at approximately the same time. This exposes such systems to an unquantifiable risk of correlated faults generating errors simultaneously in several channels. Such correlated faults might result from some external influence, such as lightning or cosmic rays, or from accumulation of latent faults not within the coverage of the diagnostics, or from design faults in the hardware logic or the software.

A much higher degree of confidence in the resilience of the system to correlated faults would result from a system design in which each channel performs its calculation at different times, on different input values, and obtains different outputs. It is even possible to consider the use of different algorithms in each of the channels. Unfortunately, as exhibited above, without an exact match between channels, standard voting techniques are vulnerable to faults that cause loss of consistency between channels and thus system failure. We seek here to provide alternative algorithms that permit differences between channels without risk of loss of consistency.

The first thoughts on an approach to such asynchronous error masking envisage a system of four channels. Each channel operates at the required iteration rate but completely unsyschronized with the other channels, thus minimizing interaction between channels. Each result produced would carry a timestamp. A processor, when voting such a result, would have access to the four most recent values, one from each channel, together with their timestamps. From these it would be possible to extrapolate to a most probable current value, as shown in Figure 25.

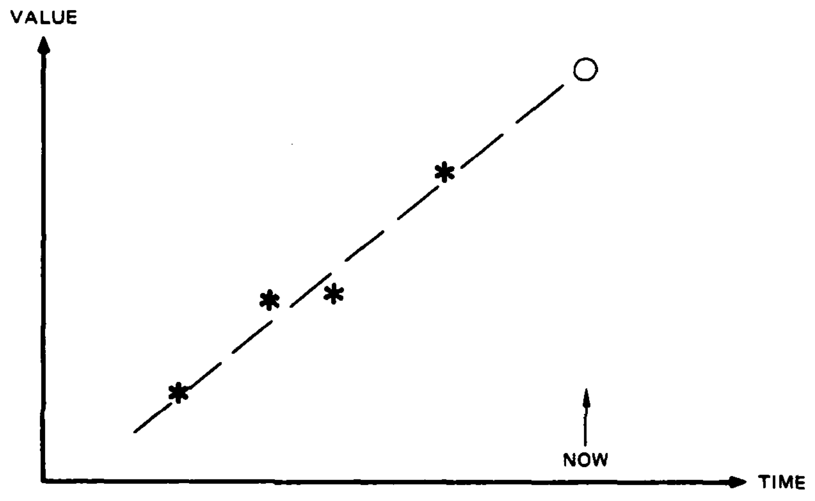


Figure 25. Extrapolation from Past Values to a Most Probable Current Value

More formally, if $R_{i,p}$ is the i'th broadcast result from processor p, containing a value $v_{i,p}$ and a timestamp $t_{i,p}$, and if the most recent result so far received from processor p is n_p, the algorithm can be expressed as:

$$\text{consensus value} = F(v_{(n_a,a)}, t_{(n_a,a)}, v_{(n_b,b)}, t_{(n_b,b)}, v_{(n_c,c)}, t_{(n_c,c)}, v_{(n_d,d)}, t_{(n_d,d)})$$

where F is some function to be determined, and a, b, c, d are the four processors.

Unfortunately, it is easy to show that the timestamps do not assist in the maintenance of consistency in the absence of any constraints on the times at which results are calculated. If greater weight is given to more recent values, those values may be erroneous values increasing the vulnerability of the system. In particular, consider the case in which three good values are reported approximately simultaneously and subsequently an erroneous value is reported. Any preference given to recent values can only render the consensus less reliable than that obtained by ignoring the timestamps.

Consideration can also be given to the clock synchronization algorithm described above. Here, if processor a is considering the values generated by processors b, c, d, with current values v_a, v_b, v_c and v_d,

$$\begin{aligned} \text{For } i \text{ in } b,c,d: v'_i = {} & \text{if } v_i > v_a + \delta \quad \vee v_i < v_a - \delta \\ & \text{then } v_a \\ & \text{else } v_i \end{aligned}$$

$$\text{and then:} \quad \text{consistent result} = \frac{v_a + v'_b + v'_c + v'_d}{4}$$

That algorithm does indeed maintain consistency between channels, but the rate of convergence is very weak and the drift and error signals that can be introduced by undetected faulty clocks are much larger than the permitted drift and jitter of working clocks. In the clock synchronization application this is not critical for the individual clocks have performance characteristics much better than those required for typical system applications. For a control system application however, the errors introduced by a faulty channel can easily overwhelm the control action of the system, and thus such an algorithm is clearly unacceptable.

A possible alternative approach requires that the four channels compute their results at uniform phases within the iteration interval, one channel generating a value at the start of the interval, a second channel generating its result a quarter of the interval later, etc., as shown in Figure 26. This additional information allows the algorithm an improved ability to compute a most probable current value and to reject erroneous values. The uniform spacing at which results are generated through the interval greatly simplifies calculations compared with a system in which such spacings are arbitrary, and thus assists in reducing the voting calculation overhead.

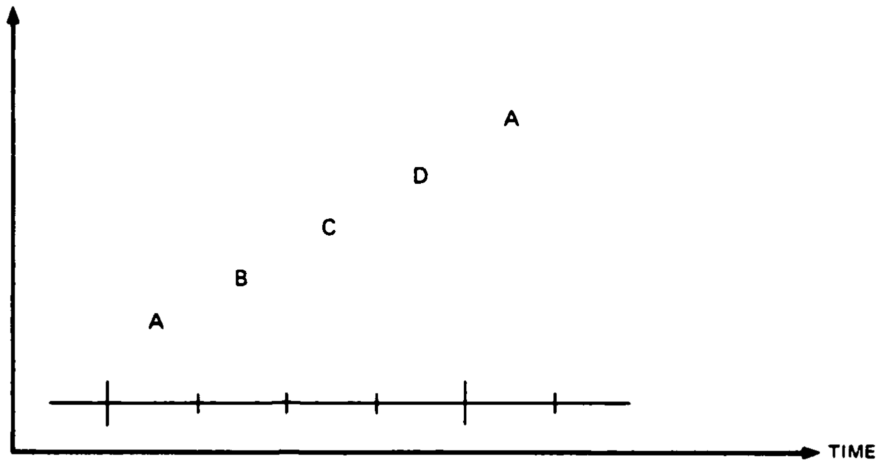


Figure 26. Calculation of Results at Uniform Phases within an Interval

An initial evaluation of such a system was made, using the arithmetic mean of the four values for the most probable current value, as in the clock synchronization algorithm. Each channel uses fixed limits for the acceptable deviation of the values computed by other channels from its own most recent value, but those limits can differ for each of the other channels. Thus if δ is an appropriate acceptable deviation for the channel whose result was computed one quarter of an iteration later, then 1.3δ is an appropriate limit for the channel computing half an iteration later and 1.2δ for the channel computing three quarters of an iteration later. These slightly larger values are permissible because the algorithm gives greater weight to more recent values, though this must be balanced against the effect of an earlier erroneous value augmenting its disturbance by influencing the intermediate values.

Here, if processor a is considering the values generated by processors b, c, d, with current values v_a, v_b, v_c and v_d,

$$\begin{aligned} v'_b = {} & \text{if } v_b > v_a + \delta \quad \vee\, v_b < v_a - \delta \\ & \text{then } v_a \\ & \text{else } v_b \end{aligned}$$

$$\begin{aligned} v'_c = {} & \text{if } v_c > v_a + 1.3\delta \quad \vee\, v_c < v_a - 1.3\delta \\ & \text{then } v_a \\ & \text{else } v_c \end{aligned}$$

$$\begin{aligned} v'_d = {} & \text{if } v_d > v_a + 1.2\delta \quad \vee\, v_d < v_a - 1.2\delta \\ & \text{then } v_a \\ & \text{else } v_d \end{aligned}$$

$$\text{and then:} \quad \text{consistent result} = \frac{v_a + v'_b + v'_c + v'_d}{4}$$

Unfortunately, while this algorithm appears to be better than the basic clock synchronization algorithm, it is only slightly so and the drift and error signals introduceable by a fault are still at least comparable to the maximum permissible control action of the system. Thus the algorithm is still unacceptable.

We can refine the algorithm by giving different weights to each of the values, for instance:

$$\text{consistent result} = \frac{v_a + 2v'_b + 3v'_c + 4v'_d}{10}$$

but the effect is marginal and still far from providing acceptable margins for control purposes.

Error masking algorithms such as these act as filters and, like all filters, necessarily introduce delay into the control loop. The algorithms above introduce a delay of about 2/3 of an iteration. To maintain the same margins of loop stability, the introduction of such a delay would require an increase in the iteration rate of about 33%.

A number of possible improvements to the algorithm are under consideration. We are currently working on algorithms that make better use of the relative timing of results, both by giving greater weight to more recent results in estimating the most probable current value, and also by considering the values generated by other channels when determining the acceptability of a result. A further possibility is the use of a five channel system fully capable of rejecting the most malicious faults which degrades on the first reconfiguration to a four channel system capable of rejecting all faults except those malicious faults in which different information is delivered to different destinations by the broadcast mechanisms. Since the probability of a second fault during a mission is low, and the probability of a malicious fault is also low, such a system might be judged to be adequately reliable.

References

[1] J. Wensley, et al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", *Proceedings of the IEEE,* 60(10):1240-1254, October 1978.

[2] A. Hopkins, T.B. Smith, J. Lala, "FTMP – A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft", *Proceedings of the IEEE,* 66(10): 1221-1239, Oct 1978.

[3] L. Lamport, P.M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults", in preparation, SRI International, Feb 1982.

[4] M. Pease, R. Shostak, L. Lamport, "Reaching Agreement in the Presence of Faults", *JACM,* 27(2):228-234, April 1980.

[5] W. H. Kautz et al., "Cellular Logic Networks and Machines", *IEEE Trans on Computers,* C-17(5):443-451, May 1968.

[6] Selim G. Akl, "Digital Signatures: Atutorial Survey", *Computer,* 16(2):15-26, February 1983.

[7] Dun, W. &Meyer, G., "A Fault Tolerant Distributed Micro-computer Structure for Aircraft Control Systems", *AIAA Guidance and Control Conference,* Palo Alto, Aug 1978.

[8] Dun, W. &Meyer, G., "Design and Analysis of a Fault Tolerant Distributed Microcomputer Control System", *IEEE Conference on Decision and Control,* San Diego, Jan 1979.